ELASTIC-BASED MULTI-SCALE GRAPH DRAWING

by

Daniel Manfred Klein

A thesis submitted to the faculty of graduate studies
Lakehead University
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Department of Computer Science

Lakehead University

2009

To my wife, without whom I would most certainly be lost

Lakehead

UNIVERSITY

OFFICE OF GRADUATE STUDIES

NAME OF STUDENT: Daniel Manfred Klein

DEGREE AWARDED: Master of Science in Computer Science

ACADEMIC UNIT: Department of Computer Science

TITLE OF THESIS: **ELASTIC-BASED MULTI-SCALE GRAPH DRAWING**

This thesis have been prepared
under my supervision
and the candidate has complied
with the Master's regulations.

_____

Signature of Supervisor

_____

Date

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I sincerely thank my thesis advisor, Dr. Maurice Benson, who has shown me more patience than I deserve and allowed me to learn how a Computer Scientist should conduct himself.

I also would like to thank Dr. Ruizhong Wei, as a member of my thesis committee and as graduate advisor for the department, for his support. I also would like to acknowledge the support and encouragement given to me by the faculty of the Department of Computer Science.

This work would not have been possible without the encouragement of my wife, Stacey, my parents, and my friends at Lakehead.

Finally, I also wish to acknowledge the valued interference provided by my kittens, as the best distraction for a busy mind is to watch other (albeit smaller ones) try to figure out the world.

Abstract

Graph drawing is an important information visualization technique with applications in a variety of disciplines, including VLSI design, bioinformatics, geography, and social network analysis. We present a new force-directed, multi-scale algorithm for the drawing of undirected graphs, using the analogy of elastics (with an unstretched length of zero), instead of springs, resulting in a force model similar to that of Tutte[65] that produces competitive results on a variety of problems.

Whereas Tutte fixes the positions of three or more vertices in order to prevent the system of elastics collapsing, we introduce pairs of vertices a fixed length apart, called 'struts' that, in part, frame existing edges, forcing the vertices along those edges apart. Also, to stiffen the system further, we add struts between vertices that are neighbours of neighbours; that is, vertices separated by a graph theoretic distance of two. Lastly, we add forces acting from the centroid of the graph vertex locations outwards to each vertex location.

We create a series of coarse-scale approximations of the graph to be drawn, employing the same algorithm used by Walshaw[68, 69]. The vertices of each graph are positioned in turn, from the coarsest to finest, with the positions of coarse graph vertices used to interpolate the initial positions of the next finer graph.

To position the vertices of each graph, we employ iterations composed of a non-linear update of the strut locations followed by a linear Gauss-Seidel update. We demonstrate the capabilities of our algorithm through a comparative study modelled after Jünger and Hachul[25] that relies on measuring the number of edge crossings in a drawing, as well as through visual comparisons. Also, to build a deeper understanding of how the various forces interact in our model, we show the results of some experiments in altering various algorithm parameters in a systematic manner.

Our new graph drawing algorithm is an effective alternate approach to traditional force-directed techniques that may be effectively used on graphs with hundreds of thousands or more vertices. Results from our algorithm are competitive with other force-directed graph drawing algorithms, producing good results for a variety of different graphs. Interestingly, our algorithm also exhibits a clustering capability superior to standard force directed methods.

Chapter 1

Introduction

The essence of graph drawing is visualizing relationships in data. This field provides important enabling technology in a variety of fields, such as diagramming, Very Large Scale Integrated (VLSI) circuit design, and the visualization of mathematical objects [64]. Graph drawing techniques have been employed in fields outside of pure computer science; for example, by bioinformaticians in the mapping of protein interactions[71, 39], sociologists in the study of social networks[51], and geographers looking either to abstract the material under study or to create a different understanding of their material by retaining relationships and discarding strict geographic location information[59]. Overviews of graph drawing in the context of social networks have also appeared in the popular press, e.g. [58]. The growing emphasis on interpreting the vast amounts of data residing in databases of all sorts currently drives and will continue to drive this area of investigation.

Automated graph layout has, directly and indirectly, been considered since 1963, when Knuth detailed an algorithm for the electronic generation of graphs mapping the control flow of computer programs[43]. His goal was to improve the documentation of software, and within the experiments he conducted, found, unexpectedly, that the diagrams began to help drive software design and debugging as well. While this idea might arguably be better considered an early ancestor of model-driven development, the graph drawing community points to this idea of using computer technology to create a visualization as having a direct influence on the history of graph drawing[4].

While much of the research into graph drawing is tied towards specific applications, as in Knuth's work, a theoretical school has developed to examine the more general questions surrounding graph visualization.

Initially, Battista, et al.[5] maintained a master list of papers on the subject which was followed by a book detailing basic applications and methods[4]. Since then, an annual conference dedicated to graph drawing as a field of study is entering, as of this writing, its fifteenth year and a website has come forward[1] to try to centrally link and collect together various resources connected with the subject.

A wide variety of techniques have been employed in graph layout, depending

on the type of graph to be visualized. Special-purpose algorithms display trees, directed graphs, and planar graphs. Perhaps one of the most broadly applied algorithms for the more general case, undirected graphs drawn with straight line edges, was first described by Eades[16]. He models the vertices and edges of a graph as a system of iron rings and springs. Since this approach can generate good results, but is computationally demanding, subsequent approaches have sought to make the computations more efficient. These iron ring and spring approaches, along with other approaches based on physical models are collectively known as force-directed methods. These approaches seek to balance forces that separate vertices in order to discern detail, and forces that bring vertices together in order to discern patterns and relationships. While this balance is difficult to achieve in all cases, force-directed approaches have continued to attract interest and further research.

The computational approaches used to solve the graph drawing problem vary widely. In Eades' algorithm[16], a variant of Hooke's Law is used for each vertex in the graph, with the vertices relocated based on the forces acting on them. Kamada and Kawai[40] take the same basic calculation and add in a relationship between the ideal length of the spring between two vertices and the graph theoretic distance between those same vertices. Springs are not just placed between existing edges, but between each vertex and every other vertex in the graph. They set the problem as an optimization exercise, defining the total energy of the entire system. The notion of defining an energy function and then minimizing it runs throughout much of the literature, e.g. [40], [5], [15], [68], [69].

Approaches to minimize the energy function are also varied. Kamada and Kawai use a block Gauss-Seidel-like approach to approximate a local minimum, calculating for one vertex while keeping all others static and then continuously repeating this process, each time selecting the vertex with the highest individual gradient for updating. Others, most notably [46, 47], actually solve a related problem – that of finding eigenvectors – to create a solution. Koren, Carmel, and Harel's approach minimizes a carefully chosen function – one "...enabling rigorous analytical analysis and straightforward implementation."[46, 47] This function, which they term *Hall's energy*[46, 47], from a Management Science paper authored in 1970[30], can be minimized efficiently with an algebraic multigrid implementation[46, 47] and the authors managed to further generalize the solution to apply to a wide variety of graphs, clustering, and partitioning problems.

Alternately, one method maps the graph edges along axes in n-dimensional space and then collapses down to two dimensions, using principle component analysis as a guide[32]. A related approach uses a similar high-dimensional process, but looks at linear combinations of vectors instead[45]. Probabilistic methods have also been employed, minimizing an energy equation through the use of simulated annealing[15, 18], as well as neural network approaches centering around Kohonen maps[50, 7, 6]. Our approach uses a multi-scale approach that avoids the complication of determining cooling schedules.

Our new approach can also be framed as an energy minimization problem. We differ from the approaches of Eades[16] and Kamada and Kawai[40] in some subtle respects. Their force-directed systems model springs with a non-zero natural length that represents a state where no potential energy is stored. When a spring is compressed to a length shorter than this, a force is exerted to restore it to its natural length. In our system, more similar to that of Tutte[41], the natural length of our springs is zero. This system would, if simulated as is, result in all vertices collapsing to a single point. Tutte[41] prevents this collapse by fixing the locations of points in the graph before performing the layout. Our method introduces a series of fixed-length segments that brace each edge of the graph. Each end point of these segments, or *struts*, connects a vertex with an additional spring. The resulting system ensures that the springs are always in some state of tension and disallows the collapse of the system into a single point. These struts introduce an additional storage cost when compared to other graph drawing algorithms, which we describe in §4.5. Consequently, our method requires neither the non-linear coupled second derivative equations of Kamada and Kawai[40], nor the inverse square global repulsion used by Eades[16]. Further, Kamada and Kawai require the calculation of the graph theoretic distance between each pair of vertices of the graph, which we also avoid.

Additionally, outward forces are introduced, applied from the centroid of the graph drawing through each vertex. These forces act to spread out all of the vertices, augmenting the local effect of the struts. Some graphs that would twist or fold into themselves can be displayed well with this addition of what we term *centroid repulsion*. However, some classes of graphs – most notably trees – are not ideally represented. We present a survey of example graphs to demonstrate the behaviour of our algorithm in §4.4.

Through our new model, we achieve an approach that compares favourably against traditional spring approaches, producing good results for a variety of different graphs. In Chapter 5, we introduce a selection of sample problems showing how our algorithm parameters interact. In the discussion that follows, these results, as well as a number of other examples, including large graphs consisting of tens of thousands of vertices, are shown. We describe where results with our algorithm compare favourably with particular example graphs given in the literature and argue that, similar to other physics-based force models, our new approach presents good results in a time efficient manner.

1.1 Notation

An undirected graph, $G = \{V, E\}$, is a set containing a set of vertices, $V = \{v_1, v_2, \ldots, v_n\}$, and a set of edges $E \subseteq V \times V$ that expresses a binary relation over V, following the notation of Koren[45]. Also as in Koren[45], we impose an order $1, \ldots, n$ on the set of vertices; thus, $n = |V|$. An edge of G between vertices v_i and v_j is given as an *unordered* pair (v_i, v_j).

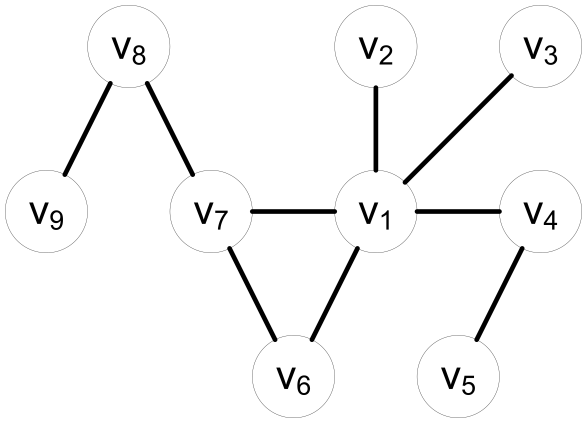


Figure 1.1: Neighbourhood function example

We consider only simply connected graphs, with no loss in generality. In the case of disconnected graphs, the various disconnected components of the graph can each be considered separately in turn.

For a two-dimensional layout of a graph G, we define a location in $\Re^2$ such that the location of vertex $v_i \in V$ is (v_i^x, v_i^y). The Euclidean distance between two vertices is given as

$$d(v_i, v_j) = \sqrt{(v_i^x - v_j^x)^2 + (v_i^y - v_j^y)^2}.$$

We also define the function $\sigma(v_i, v_j)$ as the graph theoretic or shortest-path distance from v_i to v_j[14]. That is, the function returns the number of edges that need to be traversed in the shortest path between v_i and v_j. Since we consider only connected graphs, $\sigma(v_i, v_j)$ is well-defined for any given v_i and v_j.

Of particular use in describing the Tutte model, as well as our own, is a series of neighbourhood functions, as defined by Bonabeau[6] where $N^l(v_i)$ is the $l'th$ radius neighbourhood of vertex v_i. That is, $N^0(v_i)$ is the vertex v_i itself, $N^1(v_i)$ is the set of vertices incident on v_i (that is, the neighbours of v_i), and $N^2(v_i)$ are the set of neighbours of neighbours of vertex v_i. Formally,

$$\begin{aligned}
N^0(v_i) &= \{v_i\} \\
N^1(v_i) &= \{v_j \in V \backslash N^0(v_i) \mid (v_i, v_j) \in E\} \\
N^2(v_i) &= \{v_j \in V \backslash (N^0(v_i) \cup N^1(v_i)) \mid \exists v_l \in N^1(v_i), (v_l, v_j) \in E\}
\end{aligned}$$

Consider the graph in Figure 1.1. If we examine v_1, then the sets generated by the above three functions are:

$$
\begin{aligned}
N^0(v_1) &= \{v_1\} \\
N^1(v_1) &= \{v_2, v_3, v_4, v_6, v_7\} \\
N^2(v_1) &= \{v_j \in \{v_5, v_8, v_9\} \mid \exists v_l \in \{v_2, v_3, v_4, v_6, v_7\}, (v_l, v_j) \in E\} \\
&= \{v_5, v_8\}
\end{aligned}
$$

Bonabeau[6] generalizes the neighbourhood function to $N^l(v_i)$, but for our algorithm we only require the definitions given above. These functions are used in Chapter 3 as part of our algorithm description.

The notation given in our pseudocode algorithms, e.g. Algorithm 1 in §2.2 differs in places from that of the above. For example, the locations of vertices are given in associative arrays[67, 60] such that for vertex $v_i = (v_i^x, v_i^y)$, its location is represented as $x[v_i]$ and $y[v_i]$ in the algorithm notation. A summary of the pseudocode conventions we use is presented in Table 1.1.

Algorithm	Description
$a \leftarrow b$	Assign the value of b to a
$x[v_i] \leftarrow 1.5$	Assign the value 1.5 to the x co-ordinate of vertex v_i; note that $x[v_i]$ represents v_i^x

for $i \leftarrow j, k$ **do**
⋮
end for
	Loop from j to k, inclusive, using i as the loop variable

for all $a_i \in A$ **do**
⋮
end for
	Iterate over each element a_i contained in set A

while *condition* **do**
⋮
end while
	Loop until *condition* is false

$a \leftarrow$ FUNC(b)	Call function FUNC with argument b, assigning the result to a

Table 1.1: Algorithm notation conventions

Chapter 2

Force-Directed Methods

2.1 Graph Layout is NP

General graph drawing is NP-Hard and described as the WEIGHTED GRAPH EMBEDDABILITY problem[38]. Eades, who introduced the spring method in 1984[16], makes mention of this and proposes that his heuristic physical model is a practical way to solve the problem of laying out graphs. From Johnson[38] and Eades[16] the terminology of a *spring embedder* is introduced, stemming from a set of related questions about the ability to embed graphs into other structures. In this case, WEIGHTED GRAPH EMBEDDABILITY asks if, for a weighted graph, a function exists that maps the vertices of a given graph into an n-dimensional space such that the edge lengths are equal to the edge weights[38]. Special cases of the problem can, however, result in a computationally feasible problem. For example, if vertex locations are restricted to a well-defined grid, or if the graph is a tree, polynomial-time algorithms exist[38].

Our new algorithm addresses the graph drawing problem in the same context as Eades. That is, we consider the general undirected graph drawing problem. Eades explicitly employs two criteria: that edge lengths should be as uniform as possible throughout the layout, and that the layout should be as symmetrical as possible. However, no explicit limit or tolerance is defined for the variance in edge length, nor is the concept of symmetry explicitly addressed, and we do not address symmetry in our development, as it can also be shown the symmetry is not as critical to graph readability as other measures[57, 56]. As with Kamada and Kawai's approach[40], our algorithm achieves a global balance of forces, placing this new approach within the family of force-directed graph layout algorithms.

2.2 Original Spring Method

Eades[16] proposed that the the vertices of a graph be analogous to iron rings, and the edges between the vertices springs. These springs can push vertices away from each other – a *repulsive* force, as well as draw them closer together, an *attractive*

force. Initially, Eades modelled the forces using Hooke's Law, but made two adjustments to this model in response to his experimental results. First, Eades found that the spring force between vertices far apart was too strong. Hooke's Law, $F = -kx$, where k is a spring constant and x is the distance the spring is displaced from its natural position, describes a force that depends linearly on displacement. Eades found this force to be too strong and consequently adopted a logarithmic force: $F = k \log(x/c_1)$, where c_1 is a scaling constant. Second, Eades added a force such that had non-neighbour vertices repel each other with force $c_2/\sqrt{d}$, where d is the Euclidean distance between them.

Initially, all vertices are placed randomly. Next, the force acting on each of the vertices is calculated. Finally, all of the vertices are re-positioned according to the spring forces acting on them. Eades' algorithm is described below in Algorithm 1. Note that the location of each vertex is given an element of an associative array; refer to §1.1 for further discussion on the pseudocode and graph notation.

Algorithm 1 Eades Spring Method Graph Layout Algorithm

procedure EADESLAYOUT($G, iter$)
  $x, y \leftarrow$ RANDOMPLACEMENT(G)
  for $i \leftarrow 1, iter$ **do**
    for $j \leftarrow 1, n$ **do** ▷ Calculate force on each vertex

$$\Delta_x[v_j] \leftarrow \sum_{l=1}^{n} \begin{cases} \frac{k(x[v_l]-x[v_j])\log(d(v_j,v_l)/c_1)}{d(v_j,v_l)} & (v_j, v_l) \in E \\ \frac{c_2(x[v_j]-x[v_l])}{d(v_j,v_l)^{\frac{3}{2}}} & otherwise \end{cases}$$

$$\Delta_y[v_j] \leftarrow \sum_{l=1}^{n} \begin{cases} \frac{k(y[v_l]-y[v_j])\log(c_1 \cdot d(v_j,v_l))}{d(v_j,v_l)} & (v_j, v_l) \in E \\ \frac{c_2(y[v_j]-y[v_l])}{d(v_j,v_l)^{\frac{3}{2}}} & otherwise \end{cases}$$

    end for
    for $j \leftarrow 1, n$ **do** ▷ Update vertex positions
      $x[v_j] \leftarrow x[v_j] + \rho \cdot \Delta_x[v_j]$
      $y[v_j] \leftarrow y[v_j] + \rho \cdot \Delta_y[v_j]$
    end for
  end for
end procedure

Experimentally, Eades determined that the best results were found with $k = 2$, $c_1 = 1$, $c_2 = 1$, and $\rho = 0.1$, with the last parameter being a scaling constant. Eades claims that his graph embedder works well for graphs with fewer than fifty vertices, citing performance considerations. The force calculation for all of the vertices is $O(n^2)$, but the force model Eades employs has problems other than computation time. Namely, the spring forces in more densely connected portions of the graph result in strong spring forces that tend to drive large edge length imbalances in less dense areas of some graphs[16]. Eades notes that dense graphs and graphs with weakly connected clusters are handled poorly with his method.

2.3 Adding in Graph Theoretic Distance

Building on Eades' algorithm, Kamada and Kawai[40] adopt a spring-like model, but formulate their approach quite differently. Kamada and Kawai are concerned with the overall layout of the graph and actively try to create a balanced layout. That is, they seek a graph layout that distributes the vertices and edges as uniformly as possible. To achieve this, Kamada and Kawai evaluate the force acting on each vertex and then move only the vertex with the largest magnitude force (Eades relocates all of the vertices at once). Additionally, Kamada and Kawai modify the physics model by incorporating the graph theoretic distance between pairs of vertices. Lastly, Kamada and Kawai formulate their graph drawing approach as an energy minimization problem.

Kamada and Kawai define the total energy between all vertices in the graph as:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (d(v_i, v_j) - \sigma(v_i, v_j))^2 \tag{2.1}$$

The function $\sigma(v_i, v_j)$ returns the shortest path distance between vertices v_i and v_j (see definition in §1.1). The factor k_{ij} is a stiffness factor, determining the strength of the spring between v_i and v_j. The factor is calculated through:

$$k_{ij} = \frac{K}{\sigma(v_i, v_j)^2} \tag{2.2}$$

where K is a constant. Taking this definition into consideration, the energy becomes:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{K}{2} (\frac{d(v_i, v_j)}{\sigma(v_i, v_j)} - 1)^2 \tag{2.3}$$

Necessary conditions for the minima are:

$$\frac{\partial E}{\partial x_i} = 0 \quad \frac{\partial E}{\partial y_i} = 0 \tag{2.4}$$

Unfortunately, the equations above are non-linear and coupled. Kamada and Kawai consider only one vertex at a time, treating all others as being frozen in place, solving the equations for the single vertex using a Newton method. First, for each vertex, they calculate the term Δ_i (their notation), which is the two norm of the gradient component associated with the vertex position:

$$\Delta_i = \sqrt{\left(\frac{\partial E}{\partial x_i}\right)^2 + \left(\frac{\partial E}{\partial y_i}\right)^2} \tag{2.5}$$

They choose the vertex with the largest Δ_i and update its location. After the vertex location has been updated, the Δ_i terms are all re-calculated and the next vertex

to alter is chosen. This process is repeated until the Δ_i terms reach a sufficiently small value.

Früchterman and Reingold note that "...there is much *ad hoc* argument in the literature when the authors try to explain when the algorithm terminates,"[19] referring to the algorithms of Eades[16], Kamada and Kawai[40], and Davidson and Harel[15]. That is, while termination conditions are presented, analysis on the number of iterations required to reach those conditions is not provided. Notably, in posing the question of the number of iterations of a particular force-directed layout algorithm that are needed, Früchterman and Reingold comment that there is "...little justification for the number of iterations."[19]. Arguably, determining when there are diminishing returns on increasing the number of iterations does not pose undue hardship, with good results possible over a range of termination conditions and *ad hoc* heuristics.

In order to solve the equations, the $\sigma(v_i, v_j)$ term, that is, the shortest path distance, must be calculated for all pairs $(v_i, v_j) \in V$. These values are static over the run of the algorithm, but their initial calculation is costly – requiring an All-Pairs Shortest Path (APSP) algorithm. Johnson's algorithm[14] for solving the APSP problem requires $O(|V|^2 \log |V| + |V||E|)$. The requirement of the All-Pairs Shortest Path calculation is cited as a limiting factor[31] on the Kamada-Kawai and like algorithms that incorporate graph theoretic distances, such as the multi-scale algorithm of Harel and Koren[31]. Hadany and Harel point out, however, that approximations to APSP can produce equally good layouts without incurring the additional computational cost[28, 29].

While the Kamada-Kawai algorithm is cited as being costly, the algorithm generates good results, especially for the layout of small graphs. Where the algorithm can fall into a local minimum and fail to provide a good layout for large, complex graphs, it can be effectively augmented with a multi-scale approach to overcome this limitation. The algorithms from Hadany, Harel, and Koren[28, 29, 31] are representatives of this approach where the Kamada-Kawai algorithm is incorporated as a way to refine their solutions.

2.4 Composite Algorithms

Several approaches conflate a number of independent algorithms in order to compose a more aesthetically pleasing drawing. Algorithms that use multi-scale methods, for example, are inherently composite algorithms, as they combine a coarsening algorithm in order to create a series of coarse graphs, operators for interpolation and restriction of vertices between graphs, and the algorithm to actually position the vertices. Hadany and Harel employ such an approach, creating an algorithm using the algebraic multi-grid framework[28, 29] and mention that the coarsening algorithm can critically affect the nature of the final results. Walshaw[68, 69] also uses a multi-scale approach, although more loosely than Hadany and Harel in structure and choosing a variant of Hendrickson and Leland's heuristic edge contraction

algorithm[34] to perform the graph coarsening. Walshaw's graph layout algorithm is based on that of Früchterman and Rheingold[19]. More recent work by Hu[35] proposes a more sophisticated scheme that adaptively selects between two different coarsening methods – an edge collapsing method resembling the one adopted by Walshaw[68, 69] and a maximal independent subset algorithm similar to that used by Gajer, *et al.*[21, 20]. While graph drawing algorithms seek only to apply a coarsening approach, the underlying questions of what represents an ideal coarse graph, as well as efficient means to compute it are also represented in the literature, e.g. [22].

More sophisticated graph drawing approaches combine several different algorithms into one. TopoLayout, for example, combines a multi-level algorithm with analytical components that attempt to identify specific graph features in order to produce ideal layouts for the parts of the graph with those features[3]. One multilevel graph drawing system allows for dynamic changes to the graph in real time, animating the changes to the final layout[66]. Other efforts have produced interactive systems that allow users to explore the effect of various graph drawing algorithm parameters in real time[61]. More sophisticated interactive workbench graph drawing systems, such as **Graphael**[17], **OGDF**[12], and the Graph Visualization Framework (**GVG**)[49], recognizing the composite nature of graph drawing, contain a collection of algorithms and the capabilities for both creating workflows involving multiple algorithms as well as facilities for implementing new algorithms within their respective frameworks.

Chapter 3

A New Multi-Scale Force-Directed Algorithm

Our algorithm builds on the history of force-directed algorithms, but introduces some modifications to the physical model. Initially, our energy function appears very similar to that used by Tutte[41], in that we use springs with a natural length of zero. We add two distinct components to our model that distinguishes it from Tutte – fixed length components called *struts* and a centroid repulsion force that helps to prevent folding and twisting. Also, where Tutte fixes vertex positions in order to create an effective layout, our algorithm does not and, lastly, we add a multi-scale component to our algorithm in order to better achieve a good global layout of the graph.

Hadany and Harel point out that traditional force directed and simulated annealing algorithms generally suffer as graph size increases[28]. Walshaw recognized that while the localized layout of a graph is fairly straightforward – that is, the layout of vertices within their immediate surroundings, the global untangling of a layout is more difficult[68, 69]. Früchterman and Rheingold, as well as Davidson and Harel, suggest that more complex drawings could be layed out more quickly with a multi-scale technique[15, 19]. Accordingly, both Walshaw and Hadany and Harel develop algorithms that use a series of approximations of the graph to be layed out. A successive series of less complex graphs, or, to take the terminology from multi-grid methods[11], *coarse* graphs, that have fewer vertices and edges than the original graph, are used to create a layout for the original graph.

We create a series of coarse graphs using the same method as Walshaw[68, 69], who adapted an algorithm from Hendrickson and Leland[34]. Initially, our algorithm positions the vertices of the coarsest graph randomly. Then, the force-directed component of the algorithm is iteratively applied, refining or *smoothing* the layout. Next, the vertex positions are used to interpolate the initial positions of the vertices of the next *finer* graph. To this finer graph the force-directed smoothing component is then applied. This process of interpolation and refinement is then repeated sequentially for each finer graph, similar to the multi-scale algorithm of

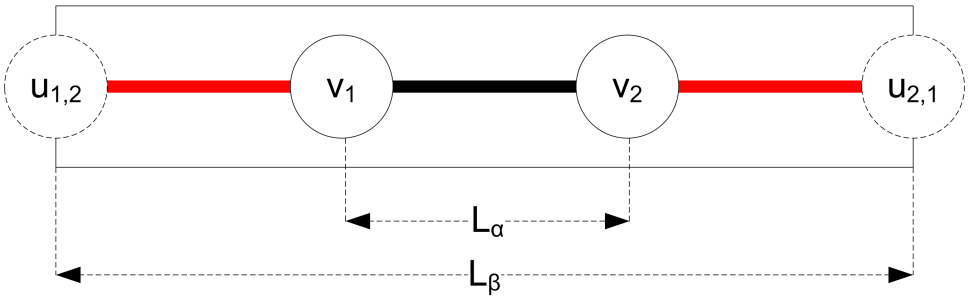


Figure 3.1: Strut model

Hadany and Harel[28]. In our, and other multi-scale algorithms, since only the coarsest graph layout starts from a random placement of vertices, artifacts such as twisting and folding that can be introduced as a consequence of initial random placement are limited.

While the multi-scale method helps to establish a good global layout, local areas of twisting or folding can still occur. To help compensate for this effect on large graphs, we introduce a repulsion factor that pushes vertices away from the centroid of the graph.

The following sections outline the various components of our algorithm, beginning with the elastic energy model, then describing the coarsening algorithm used by Walshaw, the interpolation technique, and finally combining all of the parts in a description of our multi-scale algorithm.

3.1 Elastic Model

Our method is based on "elastics", which act as springs with a natural length of zero. Figure 3.1 shows a simple graph of two vertices, v_1 and v_2. The length of the edge between the two vertices is given by L_α. With only the attractive forces between vertices, however, the model would collapse into a single point. In order to prevent this, our model introduces a fixed-length line segment positioned co-linearly with the two vertices. In Figure 3.1, this is the span L_β from $u_{1,2}$ to $u_{2,1}$. It is possible that the various forces in the overall force model acting on the vertices push them to a distance greater than L_β, in which case the fixed-length line segment, or *strut*, acts to constrain the vertices from drifting further apart.

To uniquely identify strut endpoints for each edge, strut endpoint $u_{i,j}$ is adjacent to vertex v_i and strut endpoint $u_{j,i}$ is adjacent to vertex v_j. For example, in Figure 3.1, the left strut endpoint is, therefore, $u_{1,2}$, being adjacent to v_1.

In addition to the struts bracing each edge of the graph, we introduce struts between vertices that share a common neighbour but are not neighbours themselves. Additional struts are placed between a given vertex v_i and its neighbours of neighbours as defined by Bonabeau's function, $N^2(v_i)$ (see Section 1.1). Figure 3.2

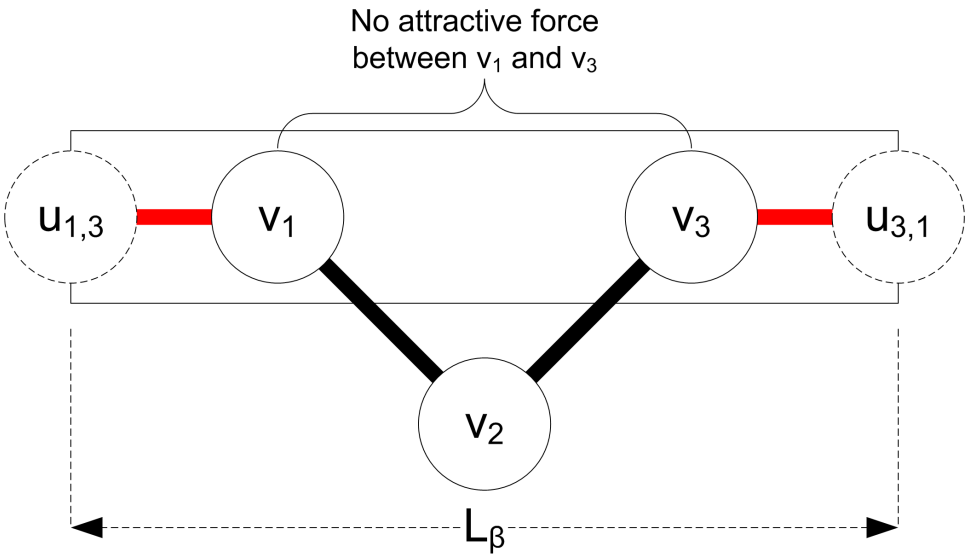


Figure 3.2: Example of a strut between non-adjacent vertices

shows a graph of three vertices, with a strut drawn using vertices v_1 and v_3. While struts along edges (such as that from Figure 3.1) have an attractive force between the vertices themselves, this is not true for the neighbour of neighbour struts. For this class of strut, attractive forces are only defined between the strut endpoints and vertices. Thus, in Figure 3.2, there are attractive forces (elastics) between v_1 and $u_{1,3}$, as well as between v_3 and $u_{3,1}$, but not between vertices v_1 and v_3.

Denote a strut along vertices v_i and v_j by the unordered pair $(u_{i,j}, u_{j,i})$. The set of all of the additional neighbour of neighbour struts is:

$$\{(u_{i,j}, u_{j,i}) \mid \exists v_i \in V, v_j \in N^2(v_i)\} \tag{3.1}$$

For example, in Figure 3.3 a graph with four vertices is shown. The strut segments along each edge are shown in red. For the additional struts, we need to apply the neighbour of neighbours function to each vertex. The results for the example are:

$$N^2(v_1) = \{v_4\} \qquad N^2(v_2) = \{v_4\}$$
$$N^2(v_3) = \emptyset \qquad N^2(v_4) = \{v_1, v_2\}$$

From this, we see that struts should be added between vertices v_1 and v_4 and vertices v_2 and v_4; these additional struts are shown in blue in the figure. On the other hand, vertex v_3 has no neighbours of neighbours, and so no additional struts are added for this vertex.

In order to discover where additional struts need to be added, the set of neighbours of neighbours for each vertex must be found. Each member of this set and the vertex forms a strut. All of the neighbour of neighbour sets can be determined

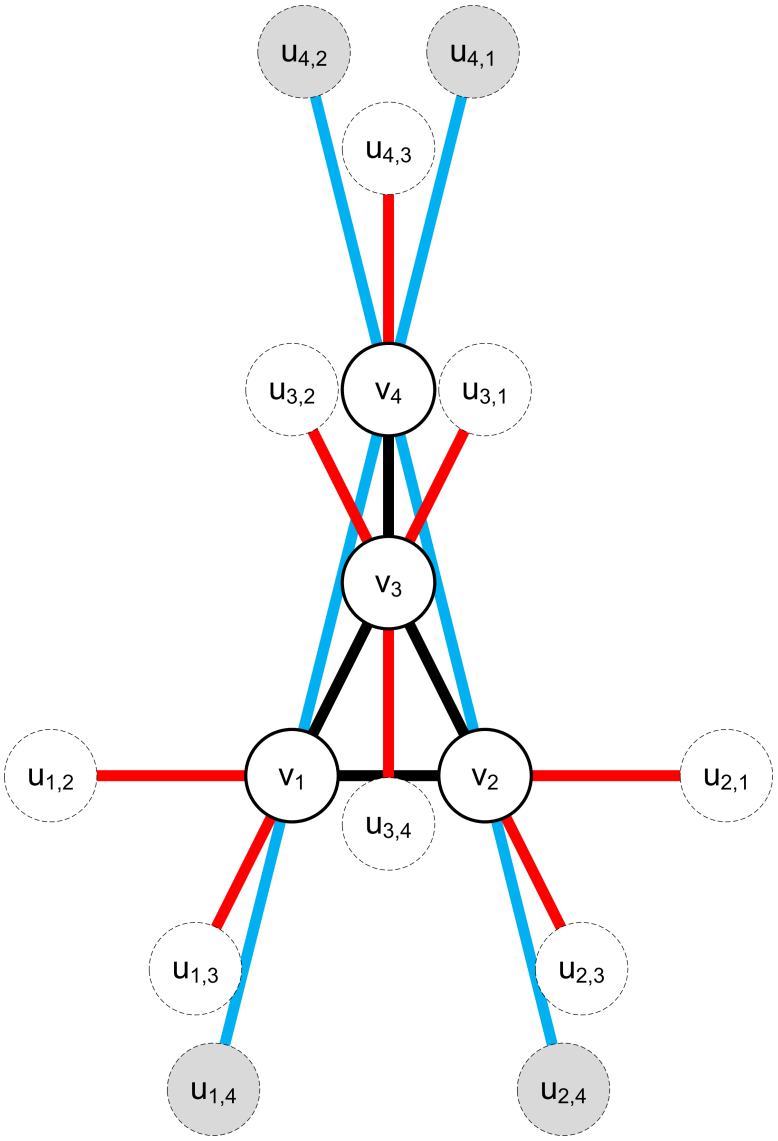


Figure 3.3: Four-vertex graph with struts

with a breadth-first walk of the graph[14], a procedure that is less computationally intensive than calculating the All-Pairs Shortest Path required by Kamada and Kawai[40] and Hadany and Harel[28] (see §2.3).

For additional control in the algorithm, the length of the struts for the neighbours of neighbours connections may be proportionately different than the struts along graph edges. By adjusting this additional parameter, significantly different results can be obtained in the layouts, as the graph can be encouraged to spread out. This parameter can also detrimentally distort the layout as well. We have found that a balance can be determined experimentally so that layouts that retain good detail and minimize twisting and folding are produced; see Chapter 5 for discussion on this and other algorithm parameters, as well as §4.4 for examples of the performance of the algorithm on graphs with various properties.

3.2 Centroid Repulsion

In order to help expand the system, as well as reduce folding and twisting, we introduce an additional force acting on each vertex. This force acts from the centroid of the graph and pulls each vertex outwards. We define a point $C = (C^x, C^y)$ as the centroid of the graph:

$$C = \left(\frac{1}{n} \sum_{i=1}^{n} v_i^x, \frac{1}{n} \sum_{i=1}^{n} v_i^y \right) \tag{3.2}$$

The centroid repulsion is modelled as an additional vertex placed at a specific length away from each vertex in the graph, co-linear with the given vertex and the centroid of the graph. We term these *centroid repulsion vertices*. For example, in Figure 3.4, a graph with four vertices is shown, with the thick black lines representing the edges of the graph. Note that $C = (C^x, C^y)$ is the centroid of the graph, as defined above. Each arrow, shown in red, is a vector from the centroid through the position of a given vertex, extending a fixed length beyond each vertex. The centroid repulsion vertex c_i in the figure, for example, will be used in the calculation for the position of vertex v_i. The vertex c_i is a fixed distance L_C away from the vertex v_i and L_C can be altered as a parameter of the algorithm. In terms of the force model, an elastic is place between c_i and v_i. Therefore, as L_C is increased, so is the force exerted by these additional elastics.

The position of a given centroid repulsion vertex c_i in the graph is calculated as:

$$c_i = \left(\frac{L_C(v_i^x - C^x)}{d(v_i, C)} + v_i^x, \frac{L_C(v_i^y - C^y)}{d(v_i, C)} + v_i^y \right) \tag{3.3}$$

If we consider the simple three vertex graph in Figure 3.5, with vertex locations

$$v_1 = (1, 1) \qquad v_2 = (4, 2) \qquad v_3 = (5, 4)$$

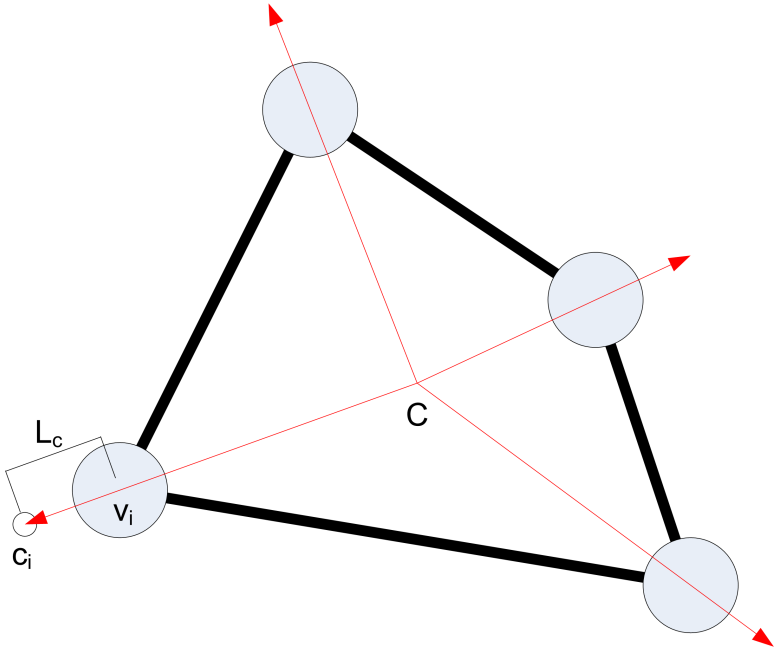


Figure 3.4: Four-vertex graph with centroid C

then the centroid of the graph is $C = (3.33, 2.33)$ and the locations of the additional centroid repulsion vertices, assuming that $L_C = 1$, are:

$$c_1 = (0.13, 0.50) \qquad c_2 = (4.80, 1.55) \qquad c_3 = (5.71, 4.71)$$

3.3 Strut Positioning

The strut end points are calculated based on the positions of the associated vertices. Since the vertex positions are initially found on the coarsest graph, and interpolated to start on the finer graphs, the strut end points can always be calculated, provided that any given pair of vertices braced by a strut do not share the same location. For two vertices $v_i, v_j \in V$ and strut end points $u_{i,j}$ and $u_{j,i}$,

$$u_{i,j} = \left(\frac{v_i^x + v_j^x}{2} + \frac{L_\beta}{2} \cdot \frac{v_i^x - v_j^x}{d(v_i, v_j)}, \frac{v_i^y + v_j^y}{2} + \frac{L_\beta}{2} \cdot \frac{v_i^y - v_j^y}{d(v_i, v_j)} \right) \tag{3.4}$$

$$u_{j,i} = \left(\frac{v_i^x + v_j^x}{2} - \frac{L_\beta}{2} \cdot \frac{v_i^x - v_j^x}{d(v_i, v_j)}, \frac{v_i^y + v_j^y}{2} - \frac{L_\beta}{2} \cdot \frac{v_i^y - v_j^y}{d(v_i, v_j)} \right) \tag{3.5}$$

For example, if we continue to consider the graph from Figure 3.5, adding the struts vertices $u_{1,2}, u_{2,1}, u_{1,3}, u_{3,1}, u_{2,3}$, and $u_{2,3}$ with $L_\beta = 6$, then the positions of the strut vertices use

$$d(v_1, v_2) = 3.16 \qquad d(v_1, v_3) = 5.00 \qquad d(v_2, v_3) = 2.24$$

and are positioned at

$$u_{1,2} = \left(\frac{1+4}{2} + \frac{6}{2} \cdot \frac{1-4}{3.16}, \frac{1+2}{2} + \frac{6}{2} \cdot \frac{1-2}{3.16}\right) = (5.35, 2.46)$$
$$u_{2,1} = \left(\frac{4+1}{2} - \frac{6}{2} \cdot \frac{4-1}{3.16}, \frac{2+1}{2} - \frac{6}{2} \cdot \frac{2-1}{3.16}\right) = (-0.35, 0.54)$$

$$u_{1,3} = (5.40, 4, 30) \qquad u_{3,1} = (0.60, 0.70)$$
$$u_{2,3} = (5.85, 5.67) \qquad u_{3,2} = (3.15, 0.33)$$

3.4 Variation on the Barycenter Method

Our analogy of elastics results in a model similar to the one used by Tutte in his barycenter method[65]. While Tutte's work is earlier, and so is not framed as a force-directed graph layout method, both di Battista, et al.[4] and Kaufman[41] re-contextualize Tutte's work in the framework of force directed graph drawing methods; we rely on their exposition in the next section.

Tutte's Barycenter method differs from those described so far in that it requires that the positions of at least three vertices be defined as part of the initialization for the algorithm. These vertices remain at their locations throughout the course of the algorithm. Using di Battista's notation, these vertices are called *fixed*, with the remaining vertices called *free*[4]. In our method, the *fixed* vertices are re-positioned after each Gauss-Seidel iteration in our linear problem, described below.

We define our *fixed* and *free* sets using the strut end points, centroid repulsion vertices, and the vertices of the graph. The *free* vertices are those of the graph G and, to include the set of *fixed* vertices, we define a graph $G' = \{V', E'\}$, first considering the struts and then considering the centroid repulsion contributions in turn. Strut end points are defined for each edge of the graph, as well as for the additional struts between neighbour of neighbour vertices. Therefore, the set V' consists of each vertex in V, all of the vertices $u_{i,j}$ for all strut end points, and all of the centroid repulsion vertices c_i:

$$\begin{aligned} V' = & V \cup \\ & \{u_{i,j}, u_{j,i} | (v_i, v_j) \in E\} \cup \\ & \{u_{i,j}, u_{j,i} | v_i \in V, v_j \in N_G^2(v_i), (v_i, v_j) \notin E\} \cup \\ & \{c_i \mid v_i \in V\} \end{aligned}$$

The edges of E' consist of the strut edges – those edges between strut end point vertices and the corresponding vertices in G – as well as the edges between the centroid repulsion vertices and the corresponding vertices in G:

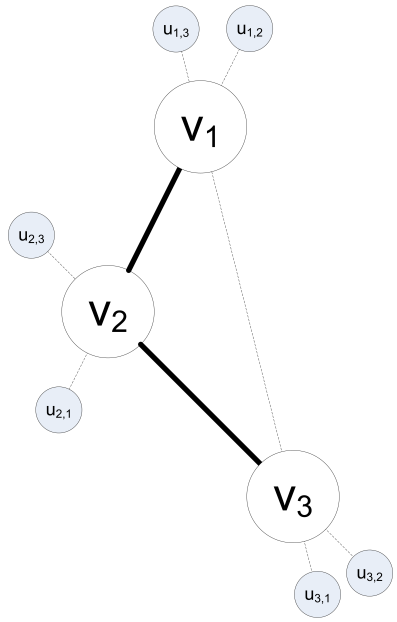


Figure 3.5: Simple three-vertex graph

$$\begin{aligned} E' =& \{(u_{i,j}, v_i), (u_{j,i}, v_j) \mid (v_i, v_j) \in E\} \cup \\ & \{(u_{i,j}, v_i), (u_{j,i}, v_j) \mid v_i \in V, v_j \in N^2(v_i), (v_i, v_j) \notin E\} \cup \\ & \{(c_i, v_i) \mid v_i \in V\} \end{aligned}$$

For example, given Figure 3.5, the set $G = (V, E)$ is:

$$\begin{aligned} V &= \{v_1, v_2, v_3\} \\ E &= \{(v_1, v_2), (v_2, v_3)\} \end{aligned}$$

There is only one additional strut, located using vertices v_1 and v_3. Given the strut endpoints, the sets V' and E', including the additional centroid vertices (not shown in diagram), are:

$$\begin{aligned} V' =& V \cup \{u_{1,3}, u_{3,1}\} \cup \{u_{1,2}, u_{2,1}, u_{2,3}, u_{3,2}\} \cup \{c_1, c_2, c_3\} \\ E' =& \{(u_{3,1}, v_3), (u_{1,3}, v_1)\} \cup \\ & \{(u_{3,2}, v_3), (u_{2,3}, v_2), (u_{1,2}, v_1), (u_{2,1}, v_2)\} \cup \\ & \{(c_1, v_1), (c_2, v_2), (c_3, v_3)\} \end{aligned}$$

Combining E and E', we define the graph $H = \{V', E \cup E'\}$.

Tutte proceeds to define, if not by name, the *Laplacian* matrix of the graph (defined and used in the context of graph drawing in [46][1]). The *Laplacian* matrix, for a simple, unweighted graph G, is $n \times n$, symmetric, and traditionally defined as $L = D - A$, where D is a diagonal matrix with entries equal to the degree of the vertex in the graph G associated with the given row under the order introduced in §1.1 and A is the adjacency matrix of the graph:

$$D = \begin{bmatrix} deg(v_1) & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & deg(v_n) \end{bmatrix} \qquad A = \begin{bmatrix} \tau(v_1, v_1) & \dots & \tau(v_1, v_n) \\ \tau(v_2, v_1) & \dots & \tau(v_2, v_n) \\ \vdots & \ddots & \vdots \\ \tau(v_n, v_1) & \dots & \tau(v_n, v_n) \end{bmatrix}$$

where

$$\tau(v_i, v_j) = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

Considering the Laplacian matrix for a given graph, the locations of the vertices satisfy the following, using our force-directed approach:

$$L \cdot x = 0 \qquad L \cdot y = 0 \tag{3.6}$$

Since these equations are singular, determining the vertex positions requires more information. Both Tutte[65] and Brandes[41] make the point that since the locations of the *fixed* vertices are defined, the sub-matrix of L created by removing the $fixed$ vertices has a non-zero determinant, implying that a unique solution to the equations exists for the co-ordinates of the $free$ vertices.

In the case of the sub-matrix defined for the *free* vertices of our matrix L, this sub-matrix is strictly diagonally dominant. That is, the sum of the absolute values of the off-diagonal elements is strictly less than the absolute value of the corresponding diagonal entry in each row. Since each edge has a strut defined along it, there is at least one $fixed$ vertex associated with each vertex in G, ensuring that the absolute value of the diagonal entry is strictly greater than the sum of the absolute values of the off-diagonal entries. Since the sub-matrix of L is symmetric, strictly diagonally dominant with positive diagonal entries, the sub-matrix is positive definite.

As noted in the beginning of this section, we solve this linear problem using the Gauss-Seidel method. While no general theory on the convergence of the Gauss-Seidel method exists[9], the following theorem from [9] is applicable:

> **Theorem.** If A is positive definite, then the Gauss-Seidel method will converge for any choice of the initial vector $x^{(0)}$.

[1] Knuth[44] also provides a definition, that provides some interesting historical context on the origin of the name *Laplacian* as well.

Given the theorem above, the Gauss-Seidel method will converge on a solution. (3.6) can be re-written for each vertex $v_i \in V$ as:

$$deg_H(v_i)v_i^x - \sum_{v_j \in N_G^1(v_i)} v_j^x = c_j^x + \sum_{\substack{j \\ u_{i,j} \in H}} u_{i,j}^x \tag{3.7}$$

$$deg_H(v_i)v_i^y - \underbrace{\sum_{v_j \in N_G^1(v_i)} v_j^y}_{free \text{ vertices}} = \underbrace{c_j^y + \sum_{\substack{j \\ u_{i,j} \in H}} u_{i,j}^y}_{fixed \text{ vertices}} \tag{3.8}$$

During an iteration, we solve for the (v_i^x, v_i^y) corresponding to the $v_i \in V$, using the temporarily fixed vertex locations in $V' \setminus V$; refer to §3.6 for a description of a full iteration. Re-arranging (3.7) and (3.8), our update formula for the position of v_i is:

$$v_i^x = \frac{1}{deg_H(v_i)} \left[\sum_{v_j \in N_G^1(v_i)} v_j^x + \sum_{\substack{j \\ u_{i,j} \in H}} u_{i,j}^x + c_i^x \right] \tag{3.9}$$

$$v_i^y = \frac{1}{deg_H(v_i)} \left[\sum_{v_j \in N_G^1(v_i)} v_j^y + \sum_{\substack{j \\ u_{i,j} \in H}} u_{i,j}^y + c_i^y \right] \tag{3.10}$$

3.5 Energy Function

The energy function calculated through the elastic forces between the edges in H resembles the various energy models presented by Koren, Carmel, and Harel[46, 47], Kamada and Kawai[40], Eades[16], Davidson and Harel[15], and Früchterman and Rheingold[19]. A useful comparative summary of these models is presented by Noack[54]. Tamassia shows how Tutte's Barycenter method can be interpreted as a force-directed graph drawing method[62].

We can use the same partitioning of vertices, given in the previous section, for the energy equations. We start with the Kamada-Kawai energy model from equation (2.1), adjusting it to correspond to our elastic model via the following:

- removing the graph theoretic distance factor
- setting the stiffness factor $k_{ij} = 1$
- calculating the energy only between vertices in H with edges using elastics with a natural length of zero

The energy for the graph H can then be defined as:

$$E = \sum_{(v_i,v_j)\in\{E\cup E'\}} \frac{1}{2} d(v_i, v_j)^2$$

Setting partial derivatives of E with respect to x and y to zero gives

$$\frac{\partial E}{\partial v_i^x} = \sum_{(v_i,v_j)\in\{E\cup E'\}} (v_i^x - v_j^x) = 0$$
$$\frac{\partial E}{\partial v_i^y} = \sum_{(v_i,v_j)\in\{E\cup E'\}} (v_i^y - v_j^y) = 0$$

resulting in a system of $2|V'|$ linear, but singular, equations. During the Gauss-Seidel part of our iteration using (3.7), (3.8), the *fixed* vertices become constraints on the energy minimization problem. Simplifying and re-arranging the terms, given the $fixed$ and $free$ vertices results in the same system given in equations (3.7) and (3.8).

3.6 Smoothing Algorithm

We use the term *smoothing* in the same sense as the multi-grid method[11], meaning that our iterations accomplish a local refinement of our graph layout solution. The two components of the smoothing algorithm that comprise a single iteration are, in order:

- **positioning the fixed vertices**: given the positions of the vertices, each strut is positioned according to the equations in §3.3 and each centroid repulsion vertex is positioned according to the equations in §3.2.
- **positioning the vertices**: given the revised position of the struts, as well as the centroid repulsion vertices, each vertex in V is re-positioned using a Gauss-Seidel iteration from equations (3.7) and (3.8).

The smoothing algorithm is presented as Algorithm 2. Note that co-ordinates are given as associative arrays[67, 60] in the algorithm notation and several components of the formulas are broken into instance variables for ease of understanding. The loop through the strut vertices $u_{i,j}$ operates with the understanding that there is a corresponding pair of vertices, v_i and v_j, which determine the strut end points.

This study employs a strategy that fixes the number of iterations on a given graph throughout the course of the layout algorithm – a strategy also used in [28, 31]. An alternate, more elaborate strategy would measure the convergence rate and cease iterating when a prescribed tolerance is met, as in, for example, Walshaw[68, 69] or using a linearly decreasing function for the number of iterations, where more iterations are performed on the coarse levels[24, 26].

Algorithm 2 Smoothing Algorithm

procedure SMOOTH(iter, $G = \{V, E\}$, $H = \{V', E \cup E'\}$)
  for $i \leftarrow 1, iter$ **do** ▷ Perform *iter* passes over the vertices
    for all $u_{i,j} \in V'$ **do** ▷ Loop through all struts
      $s_1 \leftarrow \frac{L_\beta}{2 \cdot d(v_i, v_j)}$
      $xEdgeCentre \leftarrow (x[v_i] + x[v_j])/2$
      $yEdgeCentre \leftarrow (y[v_i] + y[v_j])/2$
      $x[u_{i,j}] \leftarrow xEdgeCentre + s_1 \cdot (x[v_i] - x[v_j])$
      $y[u_{i,j}] \leftarrow yEdgeCentre + s_1 \cdot (y[v_i] - y[v_j])$
    end for
    for all $v_i \in V$ **do** ▷ Adjust positions of free vertices
      $s_2 \leftarrow \frac{1}{deg_H(v_i)}$
      $x[v_i] \leftarrow s_2(\sum_{v_j \in N^1_G(v_i)} x[v_j] + \sum_{\substack{j \\ u_{i,j} \in H}} x[u_{i,j}] + x[c_i])$
      $y[v_i] \leftarrow s_2(\sum_{v_j \in N^1_G(v_i)} y[v_j] + \sum_{\substack{j \\ u_{i,j} \in H}} y[u_{i,j}] + y[c_i])$
    end for
  end for
end procedure

3.7 Coarsening Algorithm

The goal of coarsening algorithms is to create a series of successively simpler graphs, called *coarse* graphs, that retain critical features of the more complex, or *finer* graphs. The terminology is borrowed from the multi-grid community, and Hadany and Harel introduce it in the context of graph drawing in their multi-scale algorithm[28]. Generally, coarsening algorithms contract edges to create simpler graphs and choosing which edges to contract is the central challenge. Hadany and Harel's coarsening algorithm, which uses the All-Pairs Shortest Path algorithm, chooses edges based on a cost function that performs in $O(|V|^2)$. Another coarsening algorithm originates with Harel and Koren[31], and is based on an approximation to the k-center problem – a relative of clustering that seeks to identify the graph theoretic centroids of clusters of vertices. More recently, Hachul and Jünger[24, 26] propose an alternative coarsening approach with better performance characteristics, calculating the coarse graph in $O(|V|)$ time.

We use the same coarsening algorithm as Walshaw[68, 69] in our work. This algorithm is an approximation to the maximum cardinality matching problem[68, 69]. That is, the coarsening attempts to find a *matching*, grouping all of the vertices into pairs, where possible, such that no two edges in the combined vertices are incident on the same vertex. Optimal algorithms to solve this program are available, and tuned versions of the algorithm operate in $O(|E|\sqrt{|V|}\alpha(|E|, |V|))$ time, where α is the inverse Ackerman function[22]. The graph drawing algorithm does not require an optimal matching, however. Rather, as long as a reasonably uniform

and reasonably gradual coarsening can be found, a faster algorithm approximating a maximum cardinality matching is sufficient. Walshaw uses a heuristic proposed by Hendrickson and Leland[34], for an algorithm that operates in $O(|E|)$.

The algorithm is outlined as Algorithm 3. For this algorithm, we include a set of vertex weights w such that the weight of any vertex v_i is denoted as $w[v_i]$, in associative array notation. In the Hendrickson and Leland algorithm edge weights are also computed[34], but Walshaw observed that utilizing other heuristics that preferred aggregating vertices based on edge weights did not result in better layouts[68, 69]. This routine accepts a fine graph $G_i = \{V_i, E_i, w_i\}$ and returns a coarse graph $G_{i+1} = \{V_{i+1}, E_{i+1}, w_{i+1}\}$. For the initial fine graph, each vertex has a weight of one. The COLLAPSE() subroutine referred to in the coarsening algorithm is given in Algorithm 4.

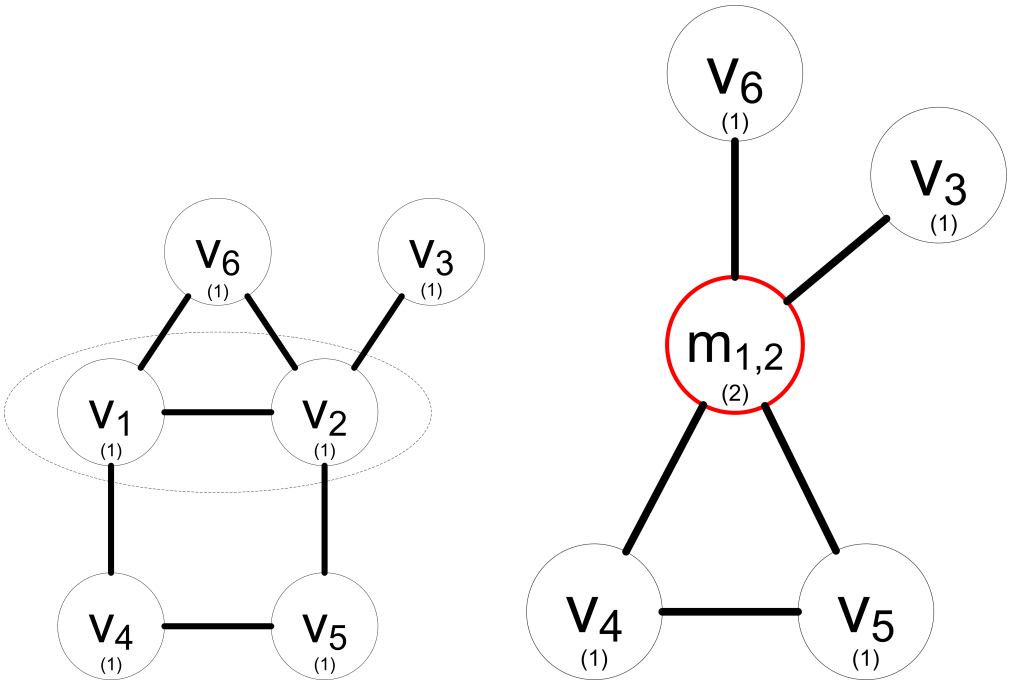
Algorithm 3 Coarsening Algorithm

```
1: function COARSEN(G_i)
2:     G_{i+1} ← G_i                          ▷ initially, G_{i+1} is a copy of G_i
3:     matched ← ∅
4:     while there remain unmatched vertices in G_{i+1} do
5:         v_m ← random unmatched vertex
6:         matched ← matched ∪ v_m
7:         v_n ← argmin{w_i[v_k] | v_k ∈ N^1_{G_i}(v_m), v_k is unmatched}
8:         if v_n ≠ ∅ then                    ▷ in case unmatched vertex is not available
9:             COLLAPSE(v_m,v_n,G_{i+1})
10:        end if
11:    end while
12:    return G_{i+1}
13: end function
```

First, a random vertex is selected from V_i and the neighbour vertex with the lowest weighted vertex that has not already been matched is selected. Where there is no single lowest weighted vertex, one is selected randomly from the set of lowest weighted vertices. For example, in Figure 3.6(a), vertices v_1 and v_2 have been matched. If v_3 is considered next, then there are no vertices with which it could be matched. In this case, v_3 becomes part of the coarse graph as is, and is neither matched or considered further in the coarsening process.

If a pair of vertices have been matched, then the COLLAPSE() subroutine removes the matched vertices and replaces them with a single coarse vertex. This process is outlined in Algorithm 4.

The intent of the vertex weighting component of the coarsening algorithm is to encourage even matching throughout the entire graph, so that vertices that may not have been aggregated in one coarse graph are more likely to be matched in subsequent applications of the algorithm. If we consider the graph in Figure 3.6(b) for coarsening, and vertex v_4 is chosen for matching, then vertex v_5 will be chosen



(a) Vertices v_1 and v_2 matched (b) v_1 and v_2 collapsed into m_{ij}

Figure 3.6: Coarsening example

over $m_{1,2}$. However, if v_6 is selected for matching, then the only candidate vertex is $m_{1,2}$; the vertex weights are not guaranteed to be uniform.

Algorithm 4 Vertex aggregation algorithm

procedure COLLAPSE(v_i, v_j, $G = \{V, E, w\}$)
  $V \leftarrow V \cup m_{i,j}$ ▷ $m_{i,j}$ is the coarse vertex
  $p \leftarrow N^1(v_i) \setminus v_j$ ▷ neighbours of v_i except matched vertex v_j
  $q \leftarrow N^1(v_j) \setminus v_i$ ▷ neighbours of v_j except matched vertex v_i
  $w[m_{i,j}] \leftarrow w[v_i] + w[v_j]$
  for all $v_k \in \{p \cup q\}$ **do**
    $E \leftarrow E \cup (m_{i,j}, v_k)$
  end for
  remove vertices v_i and v_j from V
  remove all edges incident to v_i and v_j from E
end procedure

We denote $G_1 = \{V_1, E_1\}$ as the *finest* graph. We then apply the coarsening algorithm to G_1, resulting in the definition of G_2. We continue to repeat this process until the graph $G_n = \{V_n, E_n\}$ has fewer than five vertices (that is, $|V_n| < 5$). Graphs with fewer than five vertices represent a trivial layout problem, and so no further coarsening is required. A list of all of the possible graphs with fewer than

five vertices is given in Appendix A. For each graph in the series $G_1, G_2, \ldots, G_n$, the graph is still simple and connected. Also, since at least one pair of vertices is aggregated, $|V_k| < |V_{k-1}|$, for all $k > 1$. Ideally, each vertex is matched such that $|V_{k-1}|/|V_k| = 2$, although in the worst case where only one pair of vertices is matched, the ratio is $|V_{k-1}|/(|V_{k-1}| + 1)$. The ratio of vertices between successive pairs of graphs, therefore, is in the range $(1, 2]$ using this coarsening algorithm; see §4.5 for a discussion on the best and worst case performance of the coarsening algorithm.

3.8 Multi-scale Framework

In applying our smoothing algorithm to a series of graphs, we define a multi-scale framework, an approach suggested in [19, 15], implemented originally by [28, 31, 68, 69], and used also by [35].

Algorithm 5 Multi-scale Algorithm

```
1: procedure MultiscaleLayout(iter)
2:     RandomLayout(G_n)
3:     Smooth(iter, G_n)
4:     for i ← G_n, G_{n-1}, ..., G_2 do
5:         Interpolate(G_i, G_{i-1})
6:         Smooth(iter,G_{i-1})
7:     end for
8: end procedure
```

We process a series of graphs $G_1, G_2, \ldots, G_n$, as defined above in §3.7, in turn, beginning with the coarsest graph, G_n, and proceeding to the finest graph. The framework is sketched in pseudocode as Algorithm 5. The algorithm begins with the coarsest graph, placing all of its vertices randomly. The algorithm then performs a fixed number smoothing iterations (set as a parameter of the algorithm) on the coarsest graph. Next, the positions of the vertices on the coarsest level are used to create initial positions for the next finer graph using the interpolation process shown in Algorithm 6. The process then repeats with the next finer graph, proceeding until the finest graph has had its vertices positioned.

Two forms of interpolation between levels were considered. The first form simply places the vertex or pair of vertices randomly within a circle that has a radius of one half the length of an edge – that is, $\frac{L_\alpha}{2}$ centred around the coarse vertex. We found that this strategy created too much disruption from which the Gauss-Seidel process could not compensate. A more sophisticated approach places the fine vertices according to an averaging based on the locations of the coarse vertices.

Each of the coarse vertices is considered in turn. Each coarse vertex, as a consequence of the Walshaw coarsening process, corresponds to either one or two

vertices of the fine graph. In the instance where a coarse vertex corresponds to only one fine vertex, the position of the fine vertex is assigned the location of the coarse vertex. Otherwise, where a coarse vertex corresponds to two fine vertices, an averaging process, outlined in Algorithm 6, is used to determine the positions of the fine vertices. The pseudocode contains two functions – FINE() and COARSE() that, given a vertex, return either the set of vertices that compose it in the next finer graph, or the vertex to which it is mapped onto in the next coarser graph, respectively.

If both fine vertices of a given coarse vertex share the same neighbours, they will be placed at the same location. Unfortunately, this creates a situation where the strut position cannot be determined, as there is no direction, and therefore, no vector, that can be used to create the strut end points. When this occurs, the algorithm perturbs the two vertices randomly, ensuring that they are less than a full strut length apart. The algorithm appears to be able to correct adequately for this random placement, given that in large graphs this situation occurs infrequently.

Algorithm 6 Averaging Interpolation

function AVERAGINGINTERPOLATION(G_{coarse}, G_{fine})
  for all $v_i \in G_{coarse}$ **do**
    $\{v_p, v_q\} \leftarrow$ FINE(v_i)
    if $v_q = \emptyset$ **then**
      $x[v_p] \leftarrow x[\text{COARSE}(v_i)]$
      $y[v_p] \leftarrow y[\text{COARSE}(v_i)]$
    else
      $x[v_p] \leftarrow \frac{1}{|N^1(v_p)|} \sum_{v_j \in N^1(v_p)} x[\text{COARSE}(v_j)]$
      $y[v_p] \leftarrow \frac{1}{|N^1(v_p)|} \sum_{v_j \in N^1(v_p)} y[\text{COARSE}(v_j)]$
      $x[v_q] \leftarrow \frac{1}{|N^1(v_q)|} \sum_{v_j \in N^1(v_q)} x[\text{COARSE}(v_j)]$
      $y[v_q] \leftarrow \frac{1}{|N^1(v_q)|} \sum_{v_j \in N^1(v_q)} y[\text{COARSE}(v_j)]$
    end if
  end for
end function

3.9 Interlevel Scale Factor

Consideration must be given to the length of the struts at each level. If the struts are insufficiently large, the drawing becomes tangled; too large and detail is lost as the drawing becomes stretched. Walshaw considers the question of the natural spring length in his multilevel algorithm for the same reasons we consider strut length[68, 69]. He devises a simple, but effective, scheme whereby the natural spring length for graph G_i is the product of the constant $\sqrt{\frac{4}{7}}$ and the natural spring length of the next coarser graph, G_{i+1}. Instead of determining a single constant, we choose

to make the strut length ratio between graphs a parameter of the algorithm, and discuss some of the interaction between adjusting this parameter in Chapter 5.

More precisely, our parameter defines a multiplier to the length of the struts, L_β, and we term this the *interlevel scale factor*, and denote it as ω. The length of each edge strut, therefore, for graph G_n is $\omega^n \cdot L_\beta$.

Instead of defining a fixed constant, Hu[35] uses the ratio of the diameters of the graphs G_i and G_{i+1}. Another possibility is to use the ratio of the number of vertices in successive graphs in the sequence; that is, $\omega_n = |V_{n+1}|/|V_n|$. This approach is not implemented in this study and is left for future consideration.

Through our exploration of the behaviour of our algorithm, we observed that we could produce good graph drawings by manipulating the available algorithm parameters in the context of two distinct scenarios; we

- fix the interlevel scale factor, setting it to one, and vary the strut length and centroid repulsion factor parameters, or
- fix the centroid repulsion factor, setting it to one, and vary the interlevel scale factor and strut length parameters.

In Chapter 5, we describe an experiment that explores the interaction between the various algorithm parameters in the context of the above two scenarios.

Chapter 4

Creating Good Graph Layouts

4.1 "Good" Layouts

Analysis of what constitutes a good graph layout has been the subject of conjecture since the inception of the first graph drawing algorithms and models[40]. Aesthetic comparisons are generally more opinion than fact, producing more a catalog of techniques, rather than focusing on what, precisely, is involved in creating better representations and visualizations. Some work has been done to attempt to discern precisely what affects the comprehensibility of a graph drawing, as well as establish a cognitive model that can be used to evaluate the efficacy of a particular visualization.

4.2 Criteria Affecting Graph Readability

Several criteria have been thought to affect the readability of graphs[48, 40, 55, 57, 56]. These criteria are:

- **symmetry**: the graph should be displayed with as much symmetry as possible
- **edge crossings**: overall, the number of edge crossing should be minimized
- **uniform edge length**: the length of all edges should be consistent throughout
- **edge bends**: the number of bends in edges should be minimized
- **uniform distribution**: vertices should be placed evenly throughout the drawing

Achieving edge length uniformity is NP-hard, as discussed in Chapter 2, and attempting to achieve symmetrical drawings is also difficult[16]. For that matter,

minimizing edge crossings is also NP-hard[41]. Nevertheless, some algorithms have been developed that attempt to work towards one feature or another. The randomized graph drawing algorithm from Harel and Sardas, for example, was designed with a pre-processing phase that attempts to minimize edge crossings[33] while Lipton builds a framework for graph layout based on graph symmetries[48].

The above criteria have been addressed in specific ways through planar graph drawing algorithms and, in so far as force-directed algorithms are concerned, uniform edge length is a specific side effect of the spring-based approach. With our elastic approach, we can control specifically for this by altering the L_β parameter. Uniform distribution of vertices can be approximated in space-filling neural-net algorithms[50, 6]. Since this research only considers straight-line drawings, bend minimization is not applicable to our drawings.

Although initially it was hypothesized that symmetry was crucial to the readability of graphs, and therefore a crucial part of a layout algorithm[48], in the studies conducted by Purchase[57, 56], symmetry had little effect on discerning various pieces of information from the graphs. Edge crossings, however, did play a role, and minimizing edge crossings was seen as desirable. Purchase, Cohen, and James' experiment[57] explicitly tested different graph drawing algorithms, including those of Früchterman and Reingold and Kamada and Kawai. Other algorithms tested included two planar graph drawing models, a hybrid non-deterministic model that uses a planar layout model for initial placement, a model that constructs orthogonal graph layouts, and an incremental layout that produces layouts similar to those of force-directed algorithms. The graph layouts generated from the force-directed models proved to be similar, and only the non-deterministic algorithm showed weak results. While this study does not invalidate the search for new force-directed algorithms that may perform better, it would seem to indicate that a well-constructed force-directed method ought to perform no worse than others if the images generated are at all similar.

Given the statistical evidence for the significance of edge crossings in the studies by Purchase[56, 57] on straight-line graph drawing, as well as the recent work by Hachul and Jünger[27] and an earlier study by Brandenburg, Himsolt, and Rohrer[10], this is the measure that our research will use as a basis for comparison.

4.3 Cognitive and Other studies

The origins of the readability criteria outlined in the previous section were not initially based on experimental work[36]. Research aimed towards discerning other factors involved in the readability and comprehension of graphs continues. Huang and Eades conducted an eye movement study[36] in order to attempt to determine how people read graphs. The conclusions reached did not generate any strong indicators for readability as the research was preliminary and did not have a sufficiently large sample size. Taylor and Rodgers[63] propose that concepts from the

field of graphic design could be adopted to modify graph layouts for more pleasing diagrams, specifically creating metrics for *homogeneity* and *concentration*, which measure how well a diagram utilizes the full space of the drawing area.

A further line of inquiry by Huang, Eades, and Hong[37] attempted to build a model of cognition for graph drawings. In addition to analyzing performance data, as Purchase has done, cognitive measures have been added in order to assess the effectiveness of a visualization. The experiment was task-based, and varied the complexity of the graph presented, but did not vary the technique used to layout the graph. While work such as this falls beyond the scope of this research, it is not without recognizing that the root understanding of why graph drawings are more or less effective is still elusive and, while models are being constructed to decipher how graph drawings are understood, there are still no concrete metrics that can absolutely determine what makes a "good" graph layout. While application of cognitive models has enabled the evaluation of visualizations – that is, determining whether a particular graph drawing is effective at conveying information – these models have not illuminated why a particular visualization was better or worse in doing so.

So far, the studies by Huang, *et al.* and Purchase appear to have been conducted in the traditional manner: recruit participants and have them perform certain tasks. Bovey and Rodgers[8] propose a different methodology that they hope will illuminate factors that make certain graphs easier to comprehend. They have developed a series of games, accessible through the Internet, to test such things as path finding through graphs and graph connectivity. This research is still preliminary and, while promising in concept, has not reached any specific conclusions towards desirable graph visualization properties.

While the larger question of what constitutes an effective graph visualization is being investigated, another group of researchers is attempting a narrower version: whether graph drawings are more or less effective than simply presenting the data as a table[42]. In this study, the researchers set tasks to subjects on graphs and charts of varying sizes and asked them to perform specific tasks in identifying common neighbours, shortest paths, and others. Some of their conclusions were expected – that accuracy was affected as graph edge density and the number of vertices increased, but what they did find was that in some instances, the charts – essentially, edge matrices – were just as effective as the vertex-link diagrams (that is, graph drawings). However, the study did not reveal any particular mechanism where graph drawings fail entirely in and of themselves, but did point out that for practical purposes, the value of graph layout as a representation of information is dependent on the task at hand.

4.4 Comparisons to Other Algorithms

We present a series of test graphs to both ascertain the capabilities of our algorithm, as well as compare the drawings from our algorithm with others in the literature.

Many of these graphs are chosen to test specific capabilities – maintaining symmetry, graphs with clusters, graphs with large numbers of biconnected components, and so forth. First, we present a selection of grids, a common test graph in the literature[32, 31, 28, 16, 19, 15, 21]. Next, we present a graph consisting of hexagons, contrasting the results of our algorithm with that given by Frick[18]. Lastly, Hachul and Jünger[25, 27] conducted a study of several algorithms, and we present results with our algorithm on a selection of the same graphs.

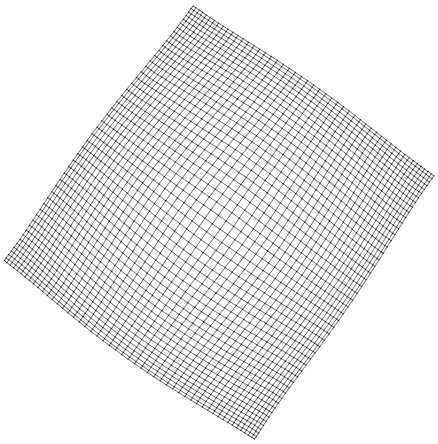
Test graphs consisting of a grid are used to demonstrate the ability of a graph drawing algorithm to maintain symmetry, as well as the ability to balance forces across regular surfaces. Sierpinski graphs are also used for this purpose, but these graphs also test how well algorithms perform where weakly connected clusters need to be kept relatively far apart.

Another common example along these lines is a grid with a certain percentage of the edges removed. In the examples shown in Figure 4.1, we begin with a 50 by 50 grid and proceed to remove, randomly, 10%, 20%, and then 30% of the edges, ensuring that the graph remains connected. Harel and Koren also present a 50 by 50 grid, both complete and with one third of the edges removed, showing that the high dimensional embedding approach[32] is capable of maintaining good overall structure, even with a substantial weakening of symmetry of the graph. In Harel and Koren's work on multi-scale algorithms[31], they also show grids with edges removed randomly, although smaller than 50 by 50. Nevertheless, Harel and Koren's multi-scale algorithm also maintains good overall structure. Results from running the same test using our algorithm, also multi-scale, shown in Figure 4.1, share this ability to maintain good overall structure.

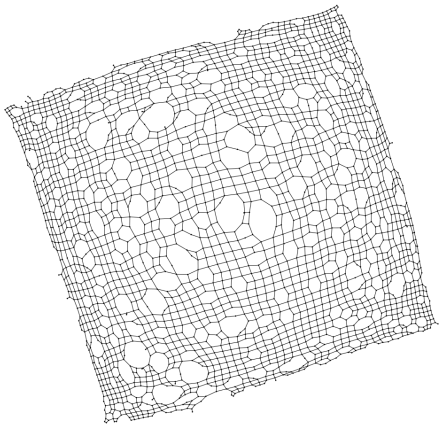
Hadany and Harel present a grid example as well, including illustrations of the various coarse graphs[28], but the results are not as consistent as those in our algorithm. Earlier work also shows grid examples[16, 19, 15, 21], but the number of vertices used is considerably smaller than 50 by 50.

Another example of our algorithm performing well on a graph with a regular structure comes in the form of an example from Frick[18]. This test graph consists of concentric rings of hexagons. A comparison between the result from Frick and our algorithm is given in Figure 4.2. The result from Frick, while still planar, does not maintain the overall hexagonal structure of the graph, distorting and flattening many of the edges; only the centre hexagon is properly displayed. In smaller examples, the Frick algorithm also distorts the shape of the hexagons.

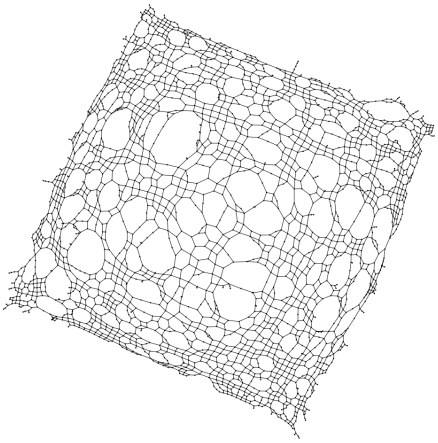
The algorithm from Frick, a simulated-annealing method calculating forces on vertices after Früchterman and Reingold[19], incorporates a gravity-like force that pulls vertices towards the centroid of the graph, the opposite of what our algorithm proposes with a repulsion force acting away from the centroid. As well, Frick's algorithm attempts to analyze oscillations and rotations in the layout, and requires more computational effort than our algorithm; while direct hardware comparisons are not possible, Frick states that, experimentally, his algorithm requires $|V|$ "rounds", where each "round" consists of $|V|$ iterations that consider all of



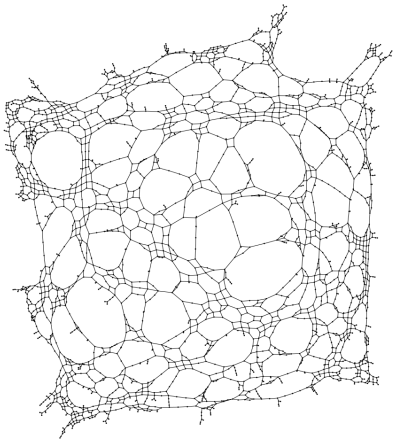
(a) no edges removed



(b) 10% of the edges removed

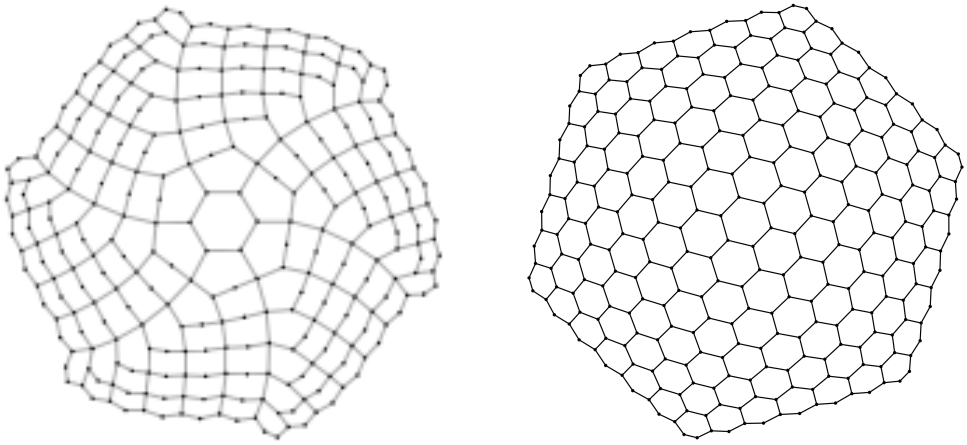


(c) 20% of the edges removed



(d) 30% of the edges removed

Figure 4.1: 50x50 grid with 0%-30% of the edges removed



(a) hexagon grid from Frick[18], reproduced with kind permission of Springer Science and Business Media

(b) hexagon grid with our algorithm

Figure 4.2: Comparing $|V| = 294$ hexagon grid results

the vertices in the graph, roughly $O(|V|^3)$ complexity. The runtime complexity of our approach varies between $O(|V|)$ and $O(|V|^4)$; see §4.5 for a discussion of the computational complexity of our algorithm.

Our algorithm is also able to generate good drawings for much larger hexagon grids, as demonstrated in Figure 4.3, where a sixty thousand vertex graph is shown. Note that the edge effects are no worse than in large square grids.

Hachul and Jünger[25, 27] conducted a study of the performance of various algorithms – the algorithm of Früchterman and Reingold[19], Gajer and Kobourov's GRIP[21], Harel and Koren's Fast Multi-Scale (FMS) Method[31] and High Dimensional Embedding (HDE) method[32], Koren's ACE[46], and Hachul's own Fast Multipole Multilevel (FM^3) method[24]. They define a metric that calculates the relative uniformity of edge length in the graph, along with a measure consisting of the ratio between the number of edge crossings and the number of edges in a graph, as well as a measure utilizing the number of overlapping edges. They provide a comparison of algorithm runtimes which, due to differing hardware, are not directly comparable to ours. For our algorithm, most examples took several seconds to run, with the graphic file generation and processing the GraphML XML files sometimes taking longer than the actual layout algorithm itself. Included in the results recorded in Table 4.2 the 'time' column records the run time of the layout algorithm, in seconds, on an Apple Macbook Pro with a 2.33Ghz Intel Core 2 Duo processor.

Arguably, edge crossings are the best measure of graph readability – see §4.2 – and so we only present that measure here. Hachul and Jünger divide the number of edge crossings by the number of edges in the graph in order to present a normalized metric; we present the same metric in Table 4.2 below.

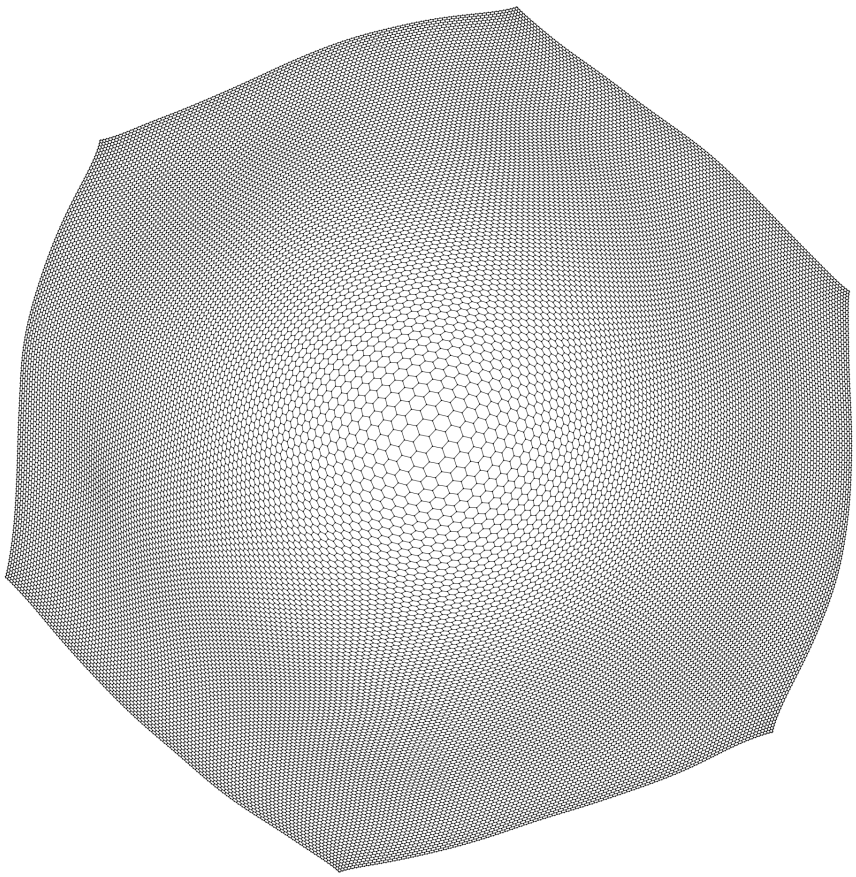


Figure 4.3: $|V| = 60000$ hexagon grid

Their selection of graphs is roughly divided into four categories: Kind Artificial, Kind Real World, Challenging Artificial, and Challenging Real World. Each of the examples was meant to address certain graph structures with which graph drawing algorithms have to contend. The "Kind" graphs tend to perform well almost universally for our and the other algorithms. The performance of the various algorithms, as well as our algorithm, varied more widely in the "Challenging" graph examples. We present a sampling from all four categories below for comparison purposes.

Two example types of Kind Artificial graphs were chosen. The first type consists of grids with three percent of their vertices removed, being careful to ensure that the graph remained connected, despite the removals. The second is a graph representation of a Sierpinski triangle. For the Challenging Artificial graphs, again, two example types were chosen. The first is a complete 6-ary tree, with four, five, or six levels. The second is what Hachul and Jünger call a *spider* graph. First, a ring containing 25% of the vertices is composed. Each vertex in the ring is connected to twelve additional adjacent vertices – six on the right and six on the left. Lastly, the remaining 75% of the vertices are divided into eight paths of roughly equal length and attached evenly along the ring. The spider graphs, therefore, contain one relatively densely connected structure along with the "legs", which are sparsely connected. The Kind Real World, as well as the Challenging Real World graphs, were downloaded from the online Walshaw collection[2], converted into GraphML files, and then used by our algorithm. The basic characteristics of each of the graphs is given in Table 4.1, including the number of vertices, edges, and the number of bi-connected components ($|B|$). A bi-connected component is a "...maximal set of edges such that any two edges in the set lie on a common simple cycle."[14]. There are some small variances in the basic graph characteristics (e.g. in the *spider* graphs) in some of our graphs and those of Hachul and Jünger, as we constructed our own graphs given their descriptions and did not have access to their graph files.

Name	$\|V\|$	$\|E\|$	$\|B\|$	$\frac{\|E\|}{\|V\|}$	max. degree
rnd_grid_032	994	1 872	6	1.9	4
rnd_grid_100	9 700	18 633	4	1.9	4
rnd_grid_320	99 328	192 075	13	1.9	4
sierpinski_06	1 095	2 187	1	2.0	4
sierpinski_08	9 843	19 683	1	2.0	4
sierpinski_10	88 575	177 147	1	2.0	4
crack	10 240	30 380	1	3.0	9
finan_512	74 753	261 120	1	3.4	54
fe_ocean	143 438	409 593	40	2.8	6
tree_06_04	1 555	1 554	1554	1.0	7
tree_06_05	9 331	9 330	9330	1.0	7

continued on next page

continued from previous page

Name	$\|V\|$	$\|E\|$	$\|B\|$	$\frac{\|E\|}{\|V\|}$	max. degree
tree_06_06	55 987	55 986	55 986	1.0	7
spider_a	994	2 494	745	2.5	15
spider_b	9 996	24 996	7497	2.5	15
spider_c	100 000	250 000	7497	2.5	15
add_32	4 960	9 462	951	1.9	31
bcsstk_31	35 586	572 914	48	16.0	188
bcsstk_33	8 738	291 583	1	33.3	140

Table 4.1: Basic properties of test graphs

We applied our algorithm to each of the graphs in turn, recording the number of edge crossings. The *relative edge crossing metric*, defined as the ratio between number of edge crossings and the number of edges in the graph was also calculated with the results summarized in Table 4.2. We ranked our results with those of the five algorithms that Hachul and Jünger tested, noting the result for each test graph in the *Rank* column of the results table (thus, our ranking is out of six, and not five algorithms). Lastly, as a more precise measure of the performance of our algorithm in comparison with the others, the *Rank %* is listed. For this value, we calculate the difference between the relative edge crossing metric produced by our algorithm for a given graph and the minimum relative edge crossing metric for all six algorithms and divide this by the difference between the maximum and minimum relative edge crossing metrics. The *Rank %* values represent, therefore, as a percentage, how far away from first place our algorithm performance was in terms of the relative edge crossing metric.

Name	time	edge crossings	$\frac{\|crossings\|}{\|E\|}$	Rank (/6)	Rank %
rnd_grid_032	0.4s	0	0.00	1 (tie)	0.00
rnd_grid_100	1.3s	0	0.00	1 (tie)	0.00
rnd_grid_320	12.7s	0	0.00	1 (tie)	0.00
sierpinski_06	0.3s	0	0.00	1 (tie)	0.00
sierpinski_08	1.2s	310	0.02	1 (tie)	0.06
sierpinski_10	5.3s	5 944	0.03	2	0.02
crack	2.4s	0	0.00	1 (tie)	0.00
finan_512	41.4s	5 197 854	19.91	3 (tie)	2.64
fe_ocean	27.8s	3 557 520	8.69	3	0.25
tree_06_04	0.5s	10 593	6.82	4	86.40
tree_06_05	1.5s	162 257	17.39	4	75.88
tree_06_06	24.2s	1 270 195	22.69	3	15.44
spider_a	0.4s	8 053	3.23	3	13.39

continued on next page

continued from previous page					
Name	time	edge crossings	$\frac{\lvert crossings \rvert}{\lvert E \rvert}$	Rank (/6)	Rank %
spider_b	0.6s	89 610	3.58	4	2.32
spider_c	6.2s	629 007	2.52	3	0.10
add_32	1.1s	54 794	5.79	4	63.16
bcsstk_31	84.0s	51 282 650	89.51	2	4.11
bcsstk_33	37.6s	116 058 775	398.03	3	7.00

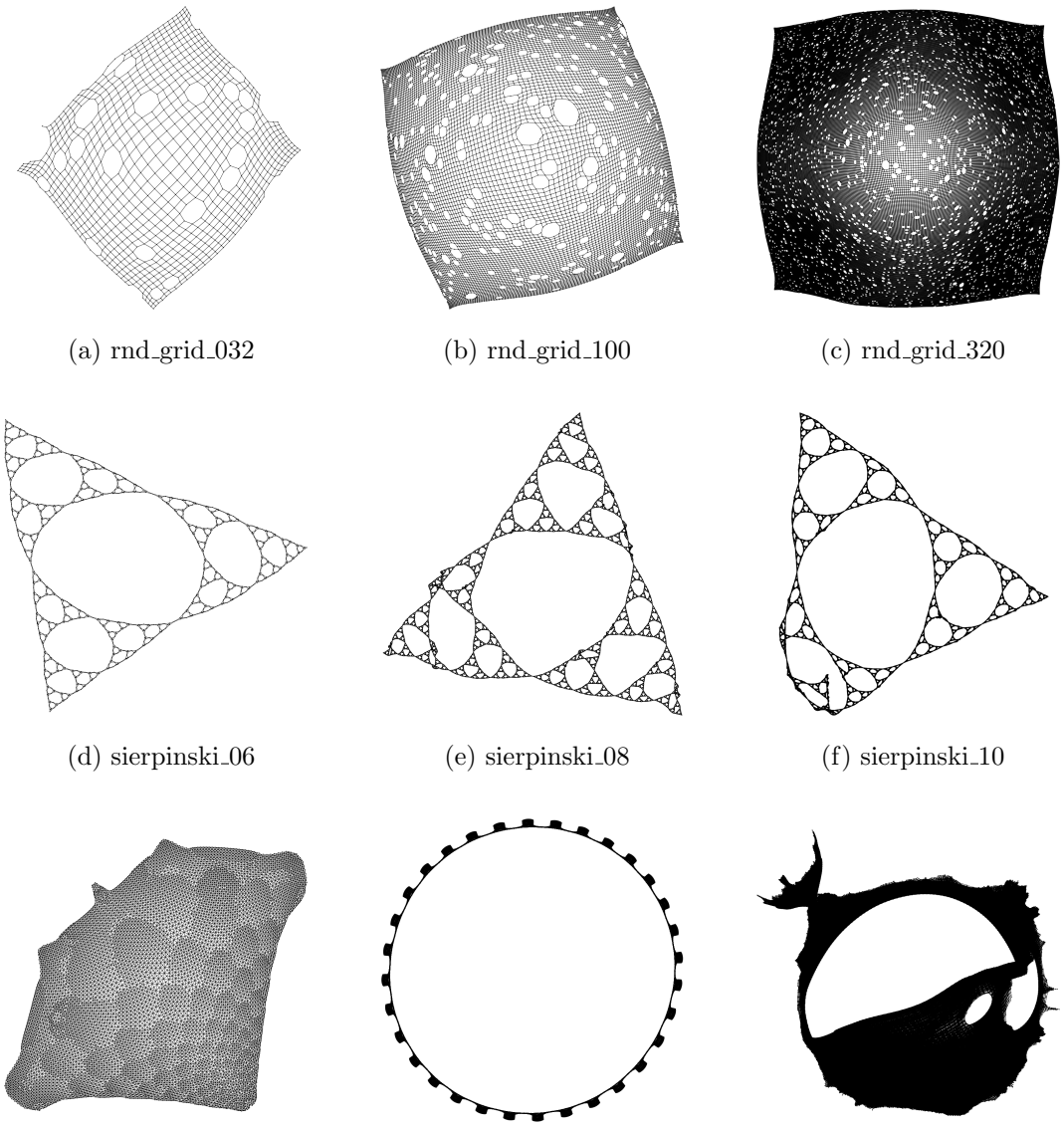
Table 4.2: Results for test graphs

The test results obtained for each graph in Table 4.2 are shown in Figures 4.4 to 4.7. Generally, the images show good detail and compare favourably to the better results from the algorithms tested by Hachul and Jünger. Specifically, the Kind Artificial graphs show good detail and no folding. Figure 4.5(a), showing the lower right portion of the 320x320 grid, clearly demonstrates a regular grid structure, despite the missing vertices. The Sierpinski graphs show some distortion and loss of visible detail; however, the structure is fairly regular, as can be seen in Figure 4.5(b), a detail of the top portion of Figure 4.4(e). The crack graph shows good, regular detail; see the zoomed section in Figure 4.5(c). The finan_512 graph shows its regular structure and while GVA and ACE do not preserve the smaller detail features very well, and the HDE algorithm fails to indicate the consistency of the structure, our result does, and is much closer to that of FM^3; see Figure 4.5(d).

The 6-ary tree proved challenging for not only the HDE and FMS methods tested by Hachul and Jünger, but for our algorithm as well. While the leaf vertices are dispersed, the root of the tree is obscured, and there is little balance or symmetry with respect to the actual tree structure. The drawings for the spider graph show the closely connected centre with each of the eight legs, although the centre ring section of the graph has much of its interconnection detail obscured; see Figure 4.7(c). The spider graphs we tested are planar, however, with ACE and HDE able to create drawings with zero edge crossings. The challenging real world graph results are comparable to the FM^3 algorithm, showing good detail and reasonably regular structure, where present.

Notably, our algorithm displays better detail than the ACE algorithm on all of the challenging real world graphs. For example, the structure of the add_32 graph is greatly obscured by ACE. Even though our rendering still has many edge crossings, there is much more detail evident, as shown in the detail of the graph in Figure 4.7(a). Another example is the bcsstk_31 graph. The ACE drawing of this graph squashes much of the corner detail on the lower right hand side of the graph, while our algorithm preserves much of this detail, as shown in Figure 4.7(b).

Our algorithm performs comparably, if not better, than the selection of other force directed algorithms evaluated by Hachul and Jünger. The results are visually similar, and the edge crossing metric is similar as well. Where graphs have regular, well-connected structure, our algorithm is able to consistently capture the overall

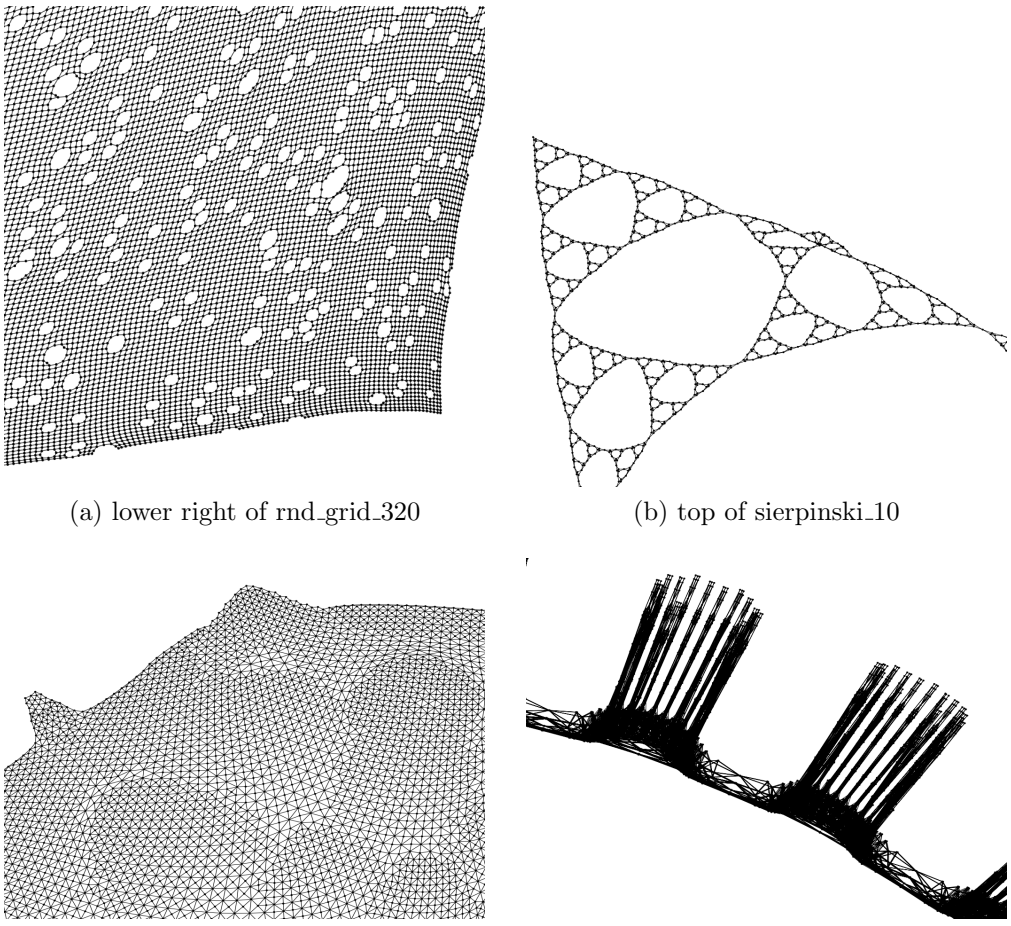


(a) rnd_grid_032 (b) rnd_grid_100 (c) rnd_grid_320

(d) sierpinski_06 (e) sierpinski_08 (f) sierpinski_10

(g) crack (h) finan_512 (i) fe_ocean

Figure 4.4: "Kind" artificial and real world graphs



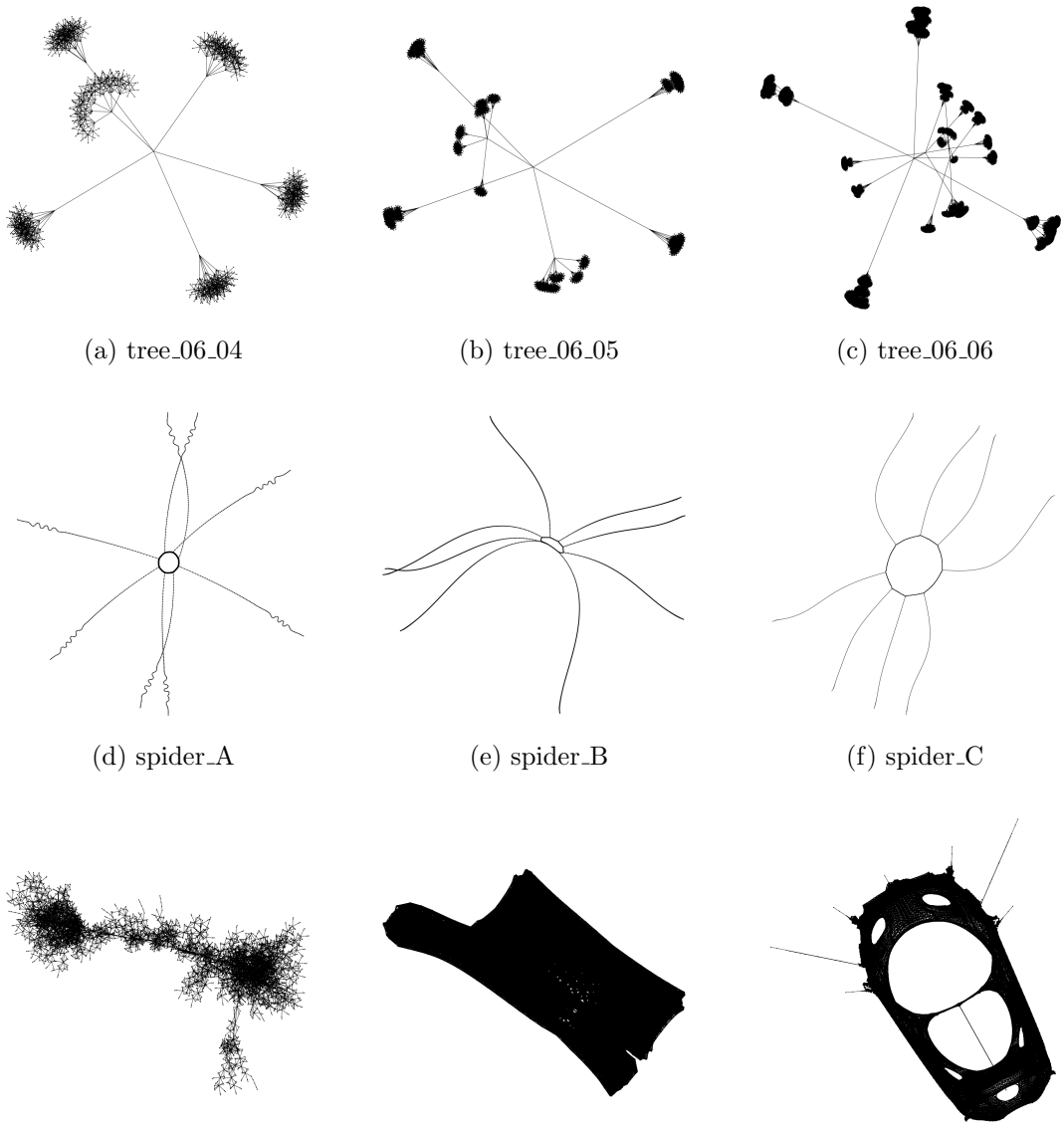
(a) lower right of rnd_grid_320

(b) top of sierpinski_10

(c) top left of crack

(d) ridge detail for finan_512

Figure 4.5: Detail views of "Kind" graphs

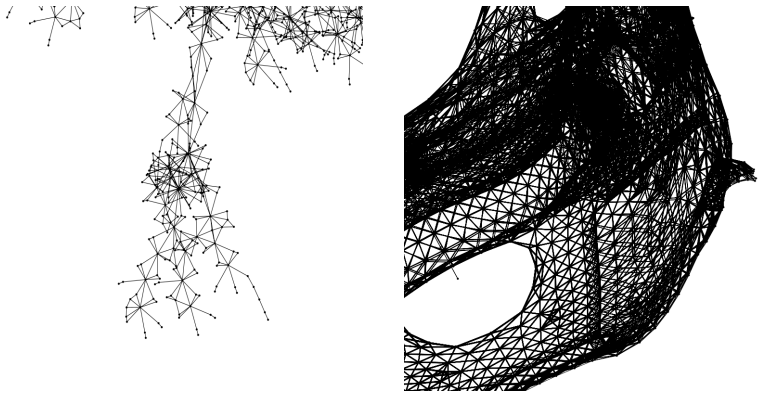


(a) tree_06_04 (b) tree_06_05 (c) tree_06_06

(d) spider_A (e) spider_B (f) spider_C

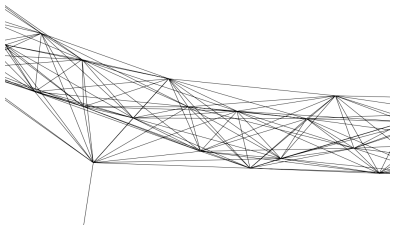
(g) add_32 (h) bcsstk_33 (i) bcsstk_31

Figure 4.6: "Challenging" artificial and real world graphs



(a) section of add_32

(b) section of bcsstk_31



(c) section of spider_a

Figure 4.7: Detail views of add_32, bcsstk_31, and spider_a

and detailed structures. For more weakly connected structures, such as trees, our algorithm does not perform as well as FM^3.

Determining the best parameter settings can be challenging for some graphs. For example, the interactions between the various parameters can result in distortions that compress the outer edges of the graph drawing, obscuring detail, as seen in the grid with a random number of edges removed, as in Figure 4.4(c), in the large hexagon grid of Figure 4.3, and particularly in Figure 5.2(c) where the outside graph edges are compressed and the central edges are pushed out from the centroid of the graph. We term this a *fisheye effect*, as it appears as if the graph is being viewed through a fisheye camera lens. Hu has identified a similar effect in Früchterman and Reingold's algorithm, calling it the *peripheral effect*[35]. By choosing low values for the centroid repulsion and by adjusting the interlevel scale factor, this effect can be mitigated, e.g. Figure 4.1, but in graphs such as the Sierpinski, attempting to remove the fisheye effect can result in a layout that twists the various parts of the graph. It is this effect, in part, that motivated the experiment in Chapter 5 to understand the interaction between the strut length, centroid repulsion, interlevel factor, and iteration parameters for the algorithm.

4.5 Algorithm Complexity

While the desire is to create a layout for the graph $G = \{V, E\}$, in order to do so we extend G with centroid repulsion and strut vertices and edges to form $H = \{V', E \cup E'\}$; see §3.4. To ascertain the computational and space complexity of the algorithm, we must consider the relationships between the number of edges and vertices of the graphs G and H. The centroid repulsion vertices add $|V|$ vertices and $|V|$ edges to V' and E', respectively. The edge struts add $2|E|$ vertices and $2|E|$ edges. Thus far, H grows linearly with respect to G. Lastly, the neighbour of neighbour struts contribute their vertices and edges, and corresponds to the sum of the $N^2(v_i)$ terms for all $v_i \in V$. In total, the number of vertices in H is:

$$|V'| = \underbrace{n}_{|V|} + \underbrace{n}_{\text{centroids}} + \underbrace{2m}_{\text{edge struts}} + \underbrace{\sum_{v_i \in V} |N^2(v_i)|}_{\text{neighbour of neighbour struts}} \tag{4.1}$$

$$= 2(n+m) + \sum_{v_i \in V} |N^2(v_i)| \tag{4.2}$$

Correspondingly, the number of edges in H is:

Graph		Vertices	Edges
Complete	G	n	$m = \frac{n(n-1)}{2}$
	H	$n^2 + n$	$3m + n$
Star	G	n	$m = n - 1$
	H	$n^2 + n$	$m^2 + 3m + 1$
Line	G	n	$m = n - 1$
	H	$6(n-1)$	$6m - 1$

Table 4.3: Vertices and edges of H in relation to G

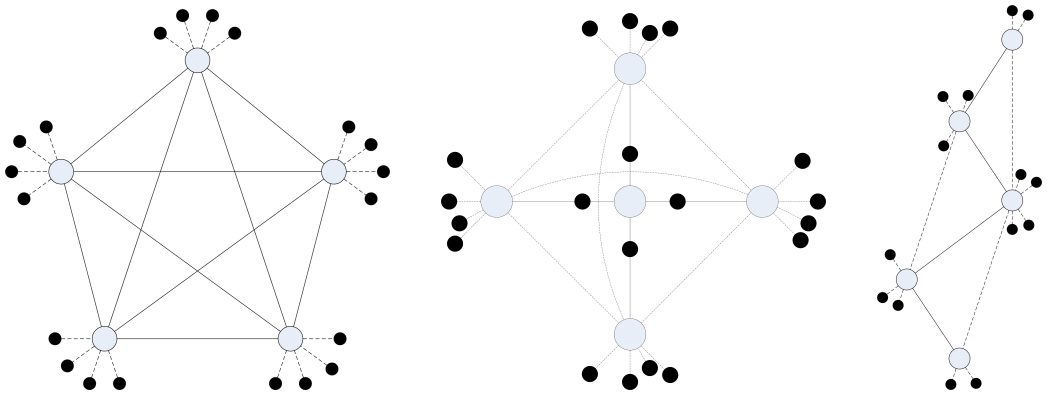
$$|E \cup E'| = \underbrace{m}_{|E|} + \underbrace{n}_{\text{centroids}} + \underbrace{2m}_{\text{edge struts}} + \underbrace{\sum_{v_i \in V} |N^2(v_i)|}_{\text{neighbour of neighbour struts}} \tag{4.3}$$

$$= n + 3m + \sum_{v_i \in V} |N^2(v_i)| \tag{4.4}$$

The number of neighbour of neighbour struts can range widely. For example, in the case where G is a complete graph, there are no neighbour of neighbour struts, and the summation is 0; see Figure 4.8(a). However, in the case of a star graph, shown with neighbour of neighbour struts drawn as dotted lines, in Figure 4.8(b), the number of neighbour of neighbour strut edges is $(n-1)(n-2)$. The star graph represents an extreme case: a 1-connected graph (that is, a graph where removing any given edge will disconnect the graph[70]) with the maximum number of possible strut edges. An in-between example, a graph with vertices arranged linearly, has fewer neighbour of neighbour strut edges: $2(n-2)$. The number of neighbour of neighbour struts determines how much larger H, in terms of the number of edges, is with respect to G. Table 4.3 summarizes these properties.

Since the number of vertices and edges in H can grow linearly or quadratically with respect to G, the storage complexity of the algorithm varies accordingly. If we assume an average case, summing the number of edges from the complete graph and the star graph and dividing by two, then the number of edges in H still exhibits $O(n^2)$ growth.

Another element to be considered is the need to store multiple levels in order to maintain the linkages needed for interpolating between graphs. Ideally, applying the coarsening algorithm we describe in §3.7 results in a coarse graph with half the number of vertices and, in the best case, this results in $\lceil \log_2 n \rceil$ graphs. A worse case, however, can be found in the star graph where an edge contraction takes place between the central vertex and one of the other vertices. Since none of the remaining vertices have common edges, no further contractions can take place, creating a sequence of $n - 1$ graphs with a bound of $O(n)$.



(a) complete graph (b) star graph (c) line graph

Figure 4.8: Example graphs showing struts

Graph Coarsening	Graph edge density	
	complete	1-connected
optimal	O(n^2)	O(n)
worst case		O(n^3)

Table 4.4: Algorithm space complexity

Considering the edge list storage requirements together with the need to store multiple graphs results in three distinct cases, summarized in Table 4.4. First, the instance where a complete graph has the best-case sequence of coarse graphs, in which case the storage requirements increase on the order of $O(n^2)$. That is, the storage required for the finest graph is $n(n-1)/2$, the storage for the next finer graph is $n(n-1)/4$, and so forth. The total storage, then, involves a geometric series: $\frac{n(n-1)}{2}[1+1/2+1/4+\cdots+1/2^n] \approx O(n^2)$. Next, if the graph is 1-connected and also can be collapsed ideally (such as the line graph, for example), then the storage requirements are better: $O(n)$, as it again involves a geometric series: $(n-1)(1+1/2+\cdots+1/2^n)$. Lastly, in the instance where the coarsening algorithm performs at its poorest, and the graph is 1-connected, then the total storage required involves a sum of squares: $f(1)+f(2)+\cdots+f(n)$ where $f(n)=(n-1)(n-2)$ and is, therefore, $O(n^3)$. In this case, the first two terms of $f(n)$ are zero, as a graph with one or two vertices does not have any additional neighbour of neighbour strut edges. The remaining possibility – that of a worst case coarsening with a complete graph is not possible, as a complete graph will allow a near perfect matching for at least the first iteration.

The runtime complexity of the algorithm is, naturally, determined by the size of graph. The graph coarsening step requires an iteration through the list of vertices and is $O(n)$. At best, creating the sequence of coarse graphs requires $\lceil \log_2 n \rceil$ applications of the coarsening algorithm and at worst $n-1$ (where $n = |V|$) ap-

plications for a runtime complexity of the coarsening process between $O(n)$ and $O(n^2)$. The best case again involves a geometric series and follows similarly as above: $n + n/2 + \cdots + n/2^{\log_2 n} \approx 2n \approx O(n)$. The worst case uses an arithmetic series giving: $n(n+1)/2 \approx O(n^2)$. Adding the struts requires a depth-first walk of the graph, an $O(n)$ operation in the best case and $O(n^2)$ in the worst. Positioning the vertices on each graph consists of a fixed number of Gauss-Seidel and strut positioning iterations. These operations operate on all of the vertices of the graph for a complexity of $O(c \cdot n)$, where c is the number of iterations. Considering the number of vertices on each level gives, as above, a best case runtime of $O(n)$ and a worst case of $O(n^2)$. For each component of the algorithm individually – coarsening, determining struts, and the graph vertex layout – the best and worst case complexities are $O(n)$ and $O(n^2)$, respectively.

However, the relationship between G and H must be considered, as $|V'|$ may be significantly larger than $|V|$. Where the coarsening process is optimal, and no additional neighbour of neighbour struts are added to the graph (e.g. a complete graph), $|V'| \approx |V|^2$, and so the algorithm runtime complexity is $O(n^2)$. In the case where the coarsening process is optimal but there are some struts (e.g. the line graph), $|V'| \approx c|V|$, and so the runtime complexity is $O(n)$. Lastly, given a worst case coarsening combined with a maximal number of struts (e.g. a star graph), then $|V'| \approx |V|^2$ and the overall runtime complexity is $n^2(1 + n^2)/2 \approx O(n^4)$. Consequently, depending on the structure of the graph – the number of added struts and the behaviour of the graph under the coarsening process – the algorithm runtime performances varies from $O(n)$ to $O(n^4)$.

Chapter 5

Algorithm Properties

5.1 Method

Several graphs were evaluated in order to discern both the efficacy of our algorithm as well as to discover how the parameters of the algorithm interact. The four main algorithm parameters are:

- **number of iterations**: the number of smoothing iterations (see §3.6) applied to each level of the graph
- **size of strut**: the length of the neighbour of neighbour struts (L_β), defined as a multiplier relative to the length of the struts enclosing edges; see §3.1
- **strength of centroid repulsion**: the factor L_C, defined as a multiplier relative to the length of the struts enclosing edges; see §3.2
- **interlevel scale factor**: the multiplier of the strut strength from a coarse to fine grid; see §3.9

An experiment was designed to test how various combinations of the above parameters interact using several different graphs. A set of values were used for the first three parameters, as given in Table 5.1. For the number of iterations parameter, the values range from performing almost no smoothing to a value where the displacement of the vertices becomes almost negligible during the last iterations. For the strut size and centroid repulsion parameters, the values again ranged from having little effect (the neighbour of neighbour struts are as strong as the edge struts) to a value where the layout should be significantly distorted. Lastly, the interlevel scale factor was held constant at one (1) for this experiment. As discussed in §3.9, we found that this set of parameters produced good graph drawings. A second experiment, corresponding to the second scenario, where the interlevel scale factor was varied and the centroid repulsion factor was held constant, is described later in this section.

Parameter	Values
number of iterations	1, 2, 5, 10, 15, 20
size of strut	1, 2, 4, 8, 16, 32
strength of centroid repulsion	1, 2, 4, 8, 16, 32

Table 5.1: Experiment parameter values

All combinations of these the parameters in Table 5.1 were tested, resulting in 216 different layout solutions for a given graph. We continue to use edge crossings as a metric for determining the quality of the layout (see §4.2 and §4.4) and calculate the number of edge crossings for each layout solution.

Another consideration is the effect of initial starting conditions and the sequence of coarsening steps on the quality of the final image. That is, since the sequence of coarse graphs generated by the Walshaw algorithm is non-deterministic, the resulting image may be affected. To attempt to mitigate the random effects, we used a set of five different sequences of coarsened graphs, each starting from a different random seed, such that the results could be averaged later.

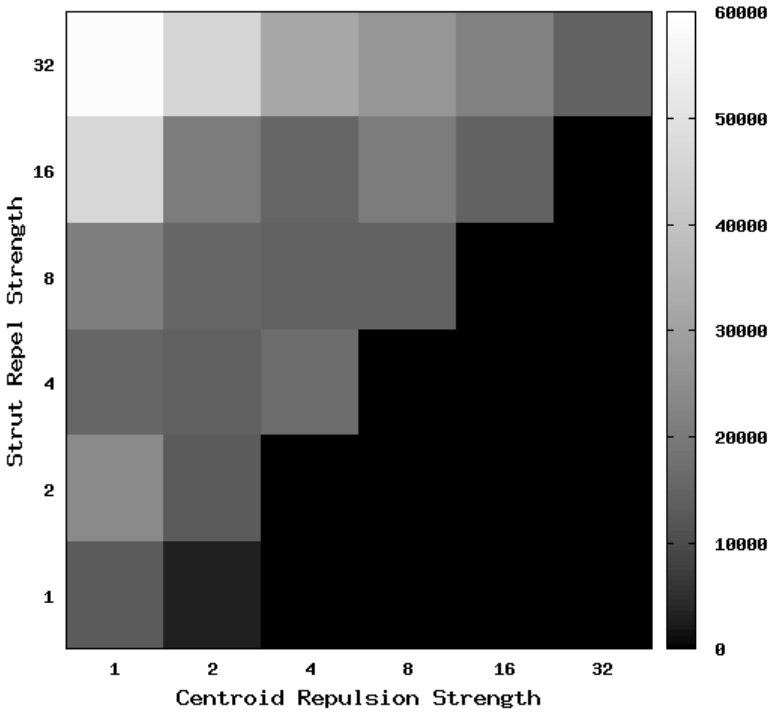
Four graphs were chosen for the above experiment. The first was a grid, 250 by 250 vertices. By far the largest graph, this graph represented a regular, well-connected surface. Second, a graph of a Sierpinski triangle, of depth 8, consisting of 3282 vertices, was used. Sierpinski graphs are also found in [21] and [35] and present a more significant challenge for a force-directed algorithm given that there are weakly connected components of the graph that must be positioned at the edges of the image. Thirdly, the *crack* graph was chosen. This graph has been used frequently in the literature as an example of a large, triangular, planar graph[3, 25, 31, 45]. Last, a random power law graph, as described by Reed[58] and Newman[51], was created using the Barabási-Albert algorithm[58]. The properties of the power law graph are mirrored in real-world systems, and as such represent an important and challenging problem for graph drawing algorithms.

5.2 Results

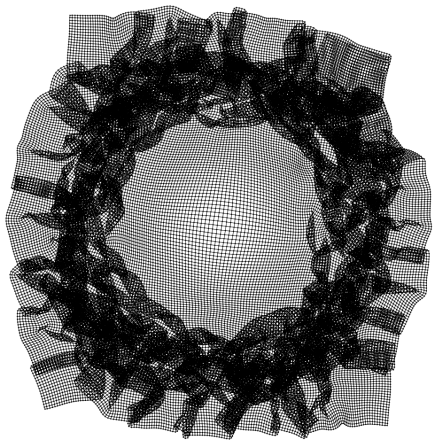
Each graph generated 216 data points (i.e. edge crossing metrics) per run and there were five runs per graph, for a total of 1080 data points per graph. In order to visualize the interaction between the various parameters, the mean of each of the five data points corresponding to the different random seeds was calculated for each of the 216 parameter combinations.

Universally, increasing the number of iterations produced better layouts. For the remainder of the discussion, we will consider the results of the experiment with the number of iterations fixed at 20, with the remaining two parameters variable.

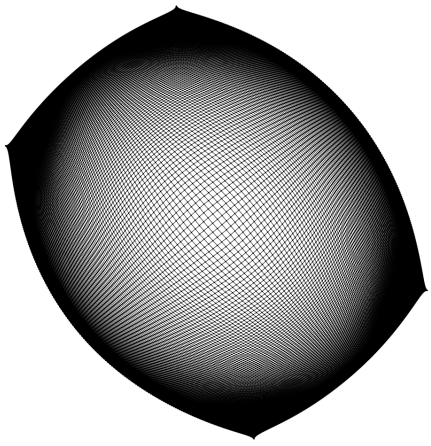
For the 250 by 250 grid, increasing the strut repulsion factor resulted in ad-



(a) 250 x 250 grid edge crossings



(b) centroid factor = 2, strut factor = 16



(c) centroid factor = 16, strut factor = 2

Figure 5.1: 250 x 250 grid test graph results

ditional edge crossings, as can be seen in Figure 5.1(a). While the edge effects become more pronounced as the centroid repulsion is increased, the edge crossings are reduced or eliminated.

The Sierpinski graph is challenging for a number of graph drawing algorithms, as well as ours. Minimizing the centroid repulsion results in a drawing that is somewhat convoluted, as can be seen in Figure 5.2(b). While strengthening the

centroid repulsion does result in fewer edge crossings, as Figure 5.2(c) shows, the amount of easily visible detail is also reduced. Arguably, a better balance between the parameters that may result in a non-optimal number of edge crossings would produce a better layout; this example suggests that the edge crossing metric is an inexact measure of graph layout readability.

A balance between the strut and centroid repulsion parameters results in the fewest edge crossings with the Sierpinski graph, whereas in the previous grid example, a bias towards a greater centroid repulsion factor and a lower strut repulsion factor results in fewer edge crossings. The algorithm parameters are sensitive to the structure of the graph and tuning the parameters can result in better layouts.

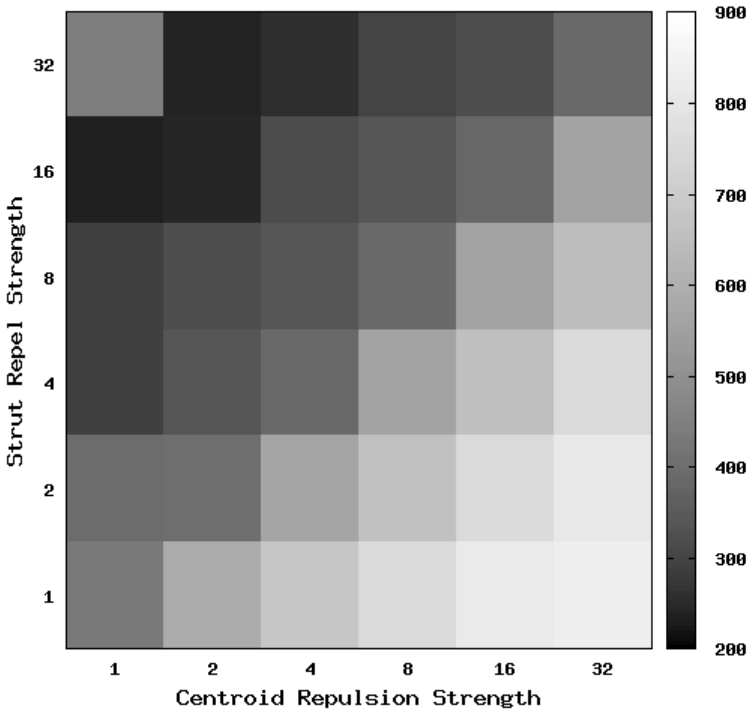
The crack graph, as aforementioned, is a common example in the graph drawing literature. This graph shows very good results with a wide variety of parameter settings. Even in the high edge crossing area at the top left of Figure 5.3(a), the results are still serviceable, as shown in Figure 5.3(b).

The results for our last test graph, the random 1000-vertex power law graph, demonstrate the challenge that real-world graphs pose to graph drawing algorithms. In this case, a poor choice for the parameters can result in a drawing with nearly no features at all; see Figure 5.4(b). However, even with a drawing with fewer edge crossings, as in Figure 5.4(c), while showing more detail, still occludes much of the structure of the graph.

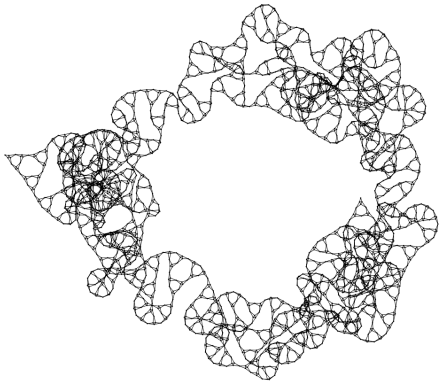
As mentioned in §3.9, we proposed two scenarios that produced good graph drawings. To explore the second scenario, the centroid repulsion factor was fixed at one (1), and we varied the strut repulsion and interlevel scaling factors. We set the interlevel parameter to a value between 1.0 and 2.0, incrementing by 0.1. The results for the 250 by 250 grid are shown in Figure 5.5. Regardless of the strut factor, increasing the interlevel scale factor results in a drawing with fewer edge crossings. The interlevel scale parameter seems have more consistent results over a larger range of values, and appears to be less sensitive than the strut factor in this example.

What we found was that the interlevel scale factor seems to act in a similar manner to the centroid repulsion vertices, although the *fisheye effect* is less pronounced. Varying the interlevel scale factor seems to have fairly profound effects on the drawing produced, suggesting that results are quite sensitive to this parameter. Both the interlevel scaling and centroid repulsion parameters cause vertices to be pushed apart, and thereby creating enough space between vertices on the coarse levels such that on the fine levels the vertices do not become tangled. These two parameters both appear to be critical in generating good graph drawings.

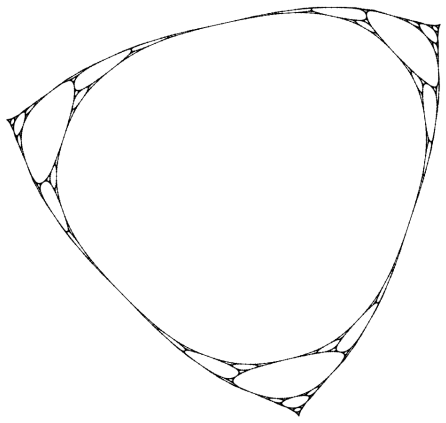
Generally, the behaviour with respect to the parameters varies widely over different graphs. We have observed no consistent choice of parameter values that performs optimally over all of the examples. However, it is clear that there are markedly better choices than others in all cases. That is, none of the examples show a uniform number of edge crossings regardless of the parameter choices. Also, as seen in the Sierpinski example, an optimal number of edge crossings may not



(a) Sierpinski graph edge crossings

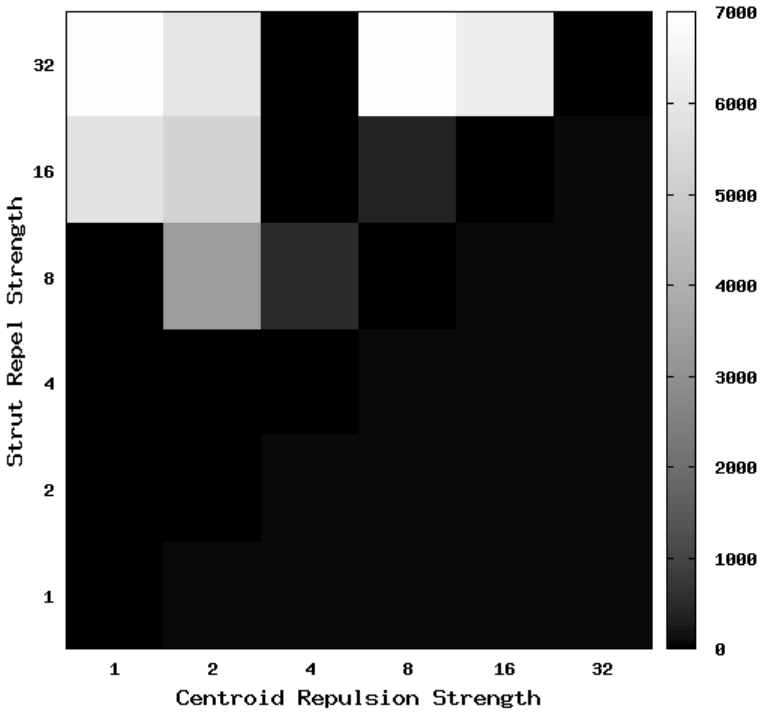


(b) centroid factor = 1, strut factor = 32

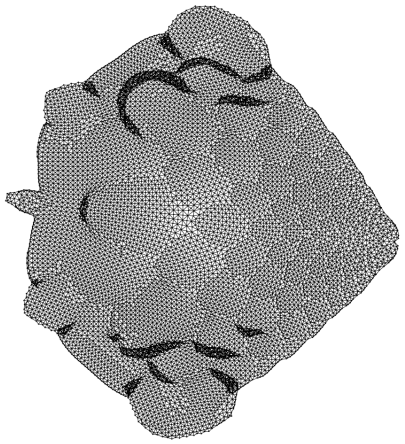


(c) centroid factor = 16, strut factor = 16

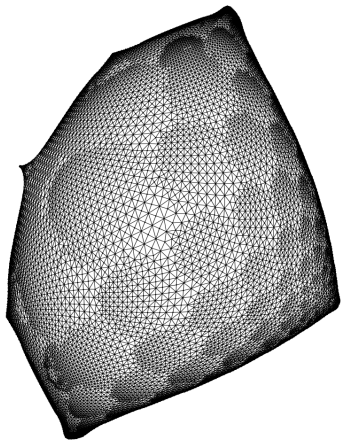
Figure 5.2: Sierpinski test graph results



(a) Crack graph edge crossings

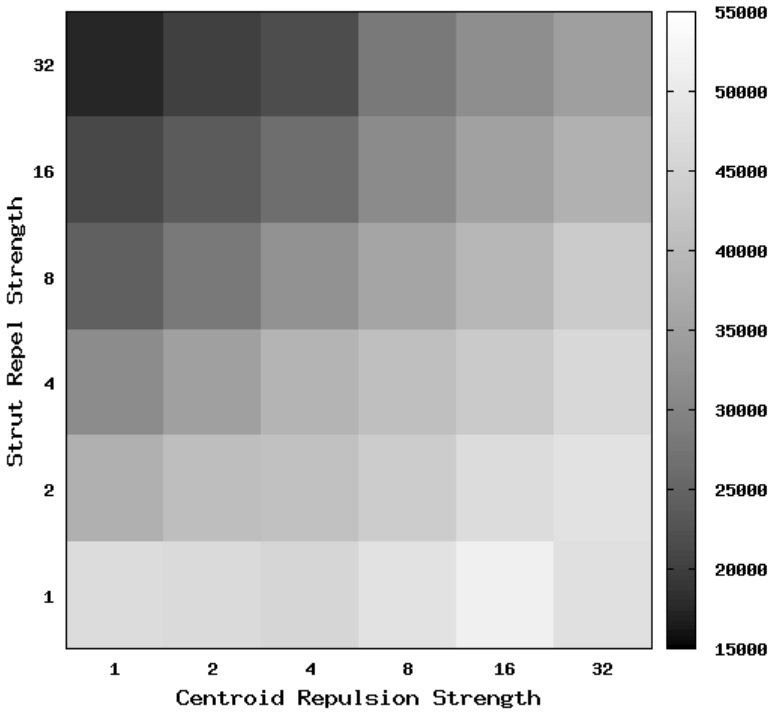


(b) centroid factor = 2, strut factor = 16

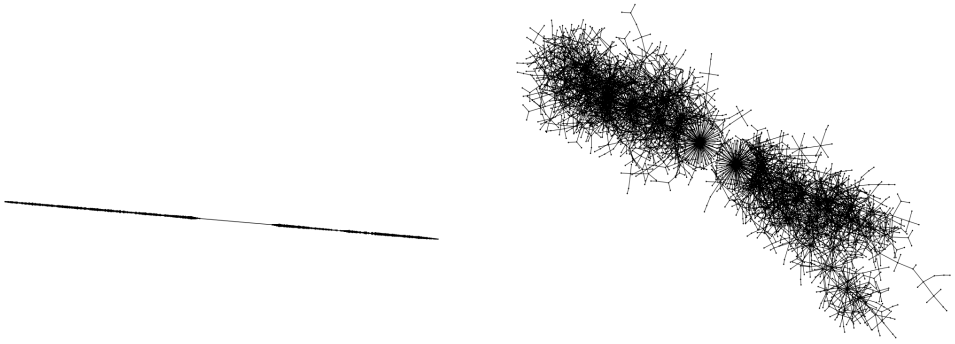


(c) centroid factor = 8, strut factor = 2

Figure 5.3: Crack test graph results

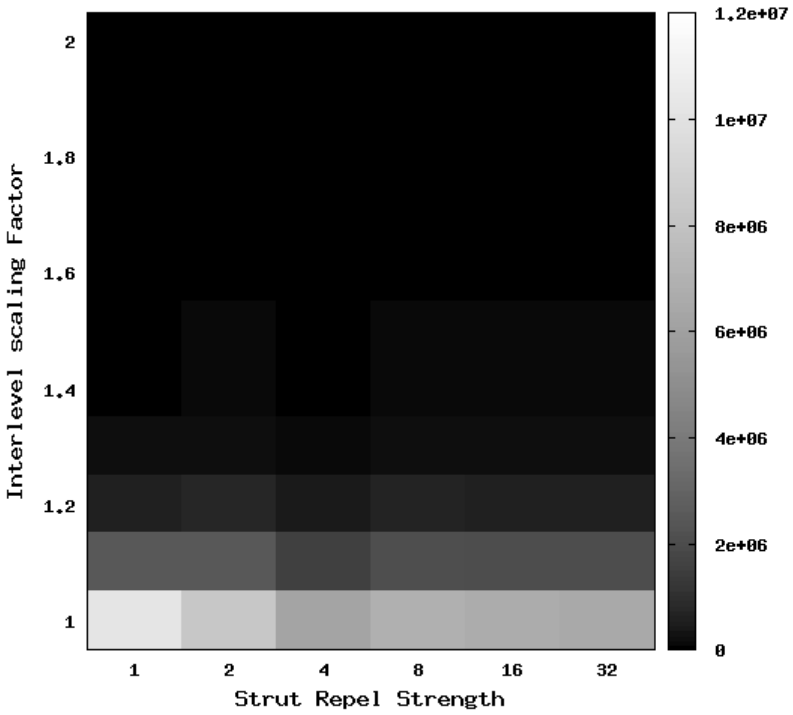


(a) Power law graph edge crossings

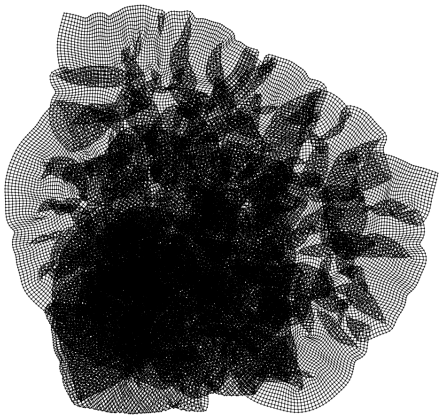


(b) centroid factor = 16, strut factor = 4 (c) centroid factor = 1, strut factor = 32

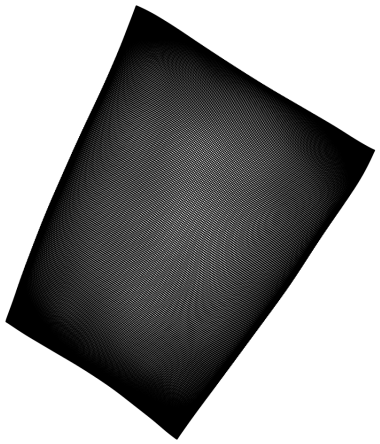
Figure 5.4: Random 1000 vertex power law test graph results



(a) 250 x 250 grid edge crossings and interlevel scaling factor



(b) strut factor = 8, interlevel scaling factor = 1.2



(c) strut factor = 8, interlevel scaling factor = 1.9

Figure 5.5: 250 x 250 grid interlevel parameter test results

produce a drawing with sufficient detail, and so judging the performance of the algorithm on edge crossings alone, while useful, is insufficient. By varying the centroid repulsion factor, strut factor, and interlevel scaling factor, very different drawings emerge. Further work in exploring the interaction environment is needed.

Chapter 6

Conclusions

We have presented a new multi-level, force-directed graph drawing algorithm that produces results comparable to those in other major algorithms. We employ the notion of elastics (with an unstretched length of zero), rather than springs, thereby creating a force model resembling that of Tutte's. However, our approach employs a unique stiffening of the force model by extending the original graph with struts and centroid repulsion vertices. In doing so, we are able to take advantage of a linear update in the positioning of the graph vertices combined with a non-linear update to the strut endpoints. At the expense of some memory, our algorithm avoids the calculation of expensive operations such as solutions to the All-Pairs Shortest Path problem. Likewise, we neither have to employ second derivatives (as in Kamada and Kawai's algorithm[40]), nor do we require an expensive $O(|V|^2)$ (or $O(|V|)$ if approximated) non-linear calculation of inverse square repulsion (that is, forces based on Hooke's Law, such as those in Eades' algorithm[16]). We combine our new force-directed approach with a multi-level approach that allows our algorithm to draw very large graphs. Since the multi-level scheme defines, with the coarse graphs, a good overall layout of the graph early, we avoid having to implement a cooling scheme, such as that used in [19, 15, 35, 69] where vertices may need to move some distance away from one another.

In presenting this algorithm, we use the comparative study of Hachul and Jünger[25, 27], employing a selection of their test problems and applying one of their quality measures. When we rank our results with those of that study, we find that our algorithm ranks in the top third nearly half the time and in the middle two thirds for the remainder. Also, we present our own study of the various parameters employed by our algorithm in order to build an understanding of their interaction. We find that there is not necessarily a setting that works optimally over all examples, necessitating some experimentation in determining the best settings.

Given the promising results for our algorithm, not only in graph drawing, but also in clustering, discussed in §6.4 below, there are several directions for future work. Broadly, further work in analyzing the parameters used in the algorithm, the coarsening process, and the computation aspects of the algorithm is warranted and

we discuss each, in turn, in the sections below.

6.1 Adaptive parameters

Currently, there are four user-configurable algorithm parameters: strut length factor, centroid repulsion factor, the number of smoothing iterations per graph, and the interlevel scale factor. We suggest that some of these parameters may be used adaptively, changing from graph to graph, throughout the course of the algorithm.

The interlevel scale factor can be effectively used to spread out the vertices in the coarse graph layouts. We suggest formulating this parameter on a given graph G_n as the ratio between the number of vertices of that graph and the preceding coarse coarse graph, G_{n+1}. That is, $\omega_n = |V_{n+1}|/|V_n|$. Since results appear to be fairly sensitive to the interlevel scale factor, and since the percentage of vertices that are coarsened on a given graph varies, this should improve how vertices are spaced and, consequently, how they are initially interpolated, thereby resulting in layouts with less folding and without loss of detail.

Another parameter that can be made adaptive is the number of smoothing iterations applied to each graph. The sum of the vertex displacements could be calculated at the completion of each iteration, and once this sum (normalized appropriately for the given coarse graph) is below a reasonable threshold, then no further iterations are performed. In this way, graph drawings that may require more work will automatically have the iterations applied on an as-needed basis.

6.2 Alternate Coarsening Algorithms

One of the main components of our multi-level algorithm is the coarsening process. For this work, we chose to use the same coarsening algorithm used by Walshaw. This algorithm, however, is not the only one possible. Hadany and Harel, for example, use a cost function to select pairs of vertices for coarsening[28]. Harel and Koren, while also using a heuristic, perform the coarsening via an approximation to the $k - center$ problem[31] while Gajer, *et al.* employ a method based on the maximal independent subset problem[20]. Hachul and Jünger employ a partitioning process, using a planetary system metaphor, that operates in $O(|V| + |E|)$ time[24] for creating coarse graphs in their multi-level algorithm. Hu, on the other hand, employs two different coarsening algorithms, selecting which to use heuristically[35].

If some properties of the graph to be drawn can be ascertained, then other coarsening algorithms that take advantage of these properties could be employed; the TopoLayout[3] system operates in this fashion. For example, if the graph is biconnected, then a coarsening algorithm based on the work of Chimani and Gutwenger, which preserves some of the non-planarity properties of the graph in the coarsening process may be employed[13].

Given the importance of the coarsening process in our algorithm, determining which algorithms would be most effective is a useful direction for this research.

6.3 Parallelization

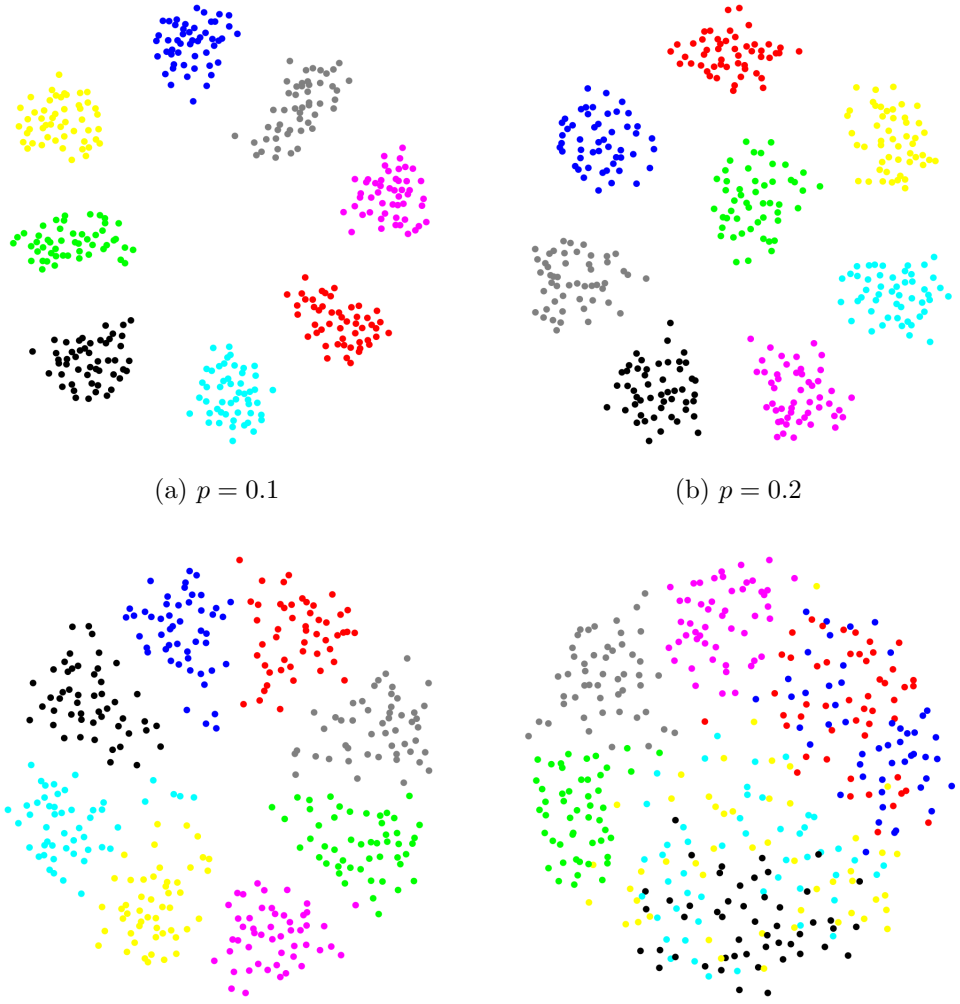
Parallelization of the smoothing operation may also be valuable. There is little literature on parallel graph drawing algorithms, and with the continuing popularity of multi-core and multi-processor systems, investigation on enhancing the performance of graph drawing algorithms in a parallel context may be useful.

6.4 Clustering

Noack[52, 53, 54] develops a graph drawing algorithm and applies it to cluster visualization. Finding clusters differs from general graph drawing in that it seeks to show subsets of vertices that are densely interconnected and have relatively few edges shared between one another. Like graph drawing, though, clustering is an important tool in software engineering, bioinformatics, and other fields[52].

Noack states that many popular force-directed graph drawing algorithms do not perform clustering well[52]. However, while our force-direct algorithm was not specifically designed for the purpose of clustering, we believe that our algorithm can produce good results in these sorts of problems as well. As a test, Noack uses an example graph from Garbers, *et al.*[23]. Noack denotes this graph as $G(k, n, p_{int}, p_{ext})$, where k is the number of clusters, n is number of vertices in each cluster, p_{int} is the probability that an edge between two vertices in the same cluster exists and p_{ext} is the probability that an edge between two vertices of different clusters exists. We replicate Noack's example from [52] in Figure 6.1.

In this figure, we define a pseudo-random graph with eight clusters of fifty vertices each. All of the vertices within a given cluster have edges to each other vertex in the cluster and the probability that an edge between vertices of different clusters exists is varied between 10% to 60%. As the number of edges between clusters increases, the clusters become indistinct. Our algorithm, without modification, is able to clearly separate out the clusters present in the randomly generated example graphs. The results are encouraging and mirror those from Noack's LinLog model, which is explicitly designed for cluster visualization. Consequently, we believe that a valuable line of inquiry is the application of our algorithm to clustering problems.



(a) $p = 0.1$ (b) $p = 0.2$

(c) $p = 0.4$ (d) $p = 0.6$

Figure 6.1: Random graph clustering example: $G(8, 50, 1.0, p)$

Appendix A

Simple Graphs

The coarsening algorithm we describe in §3.7 ceases when the number of vertices in the coarse graph is fewer than five. All of the graphs with four or fewer vertices are, due both to their simplicity, as well as the additional stiffness provided by the struts, stable. That is, regardless of initial position, these graphs quickly attain the same configuration.

These stable graphs are enumerated in the tables below. Note that a graph of one vertex requires no layout, as it may be placed arbitrarily, and as such represents a trivial case.

A.1 Two Vertex Graph

There is only one configuration with two vertices. This case is also trivial, with only one edge and no additional struts; see Figure A.1. While the illustrations of these graphs is somewhat arbitrary and dependent on initial random placement, the configurations are invariant on rotation and on translation. That is, regardless of the initial random starting position of the vertices in a 2, 3 or 4 vertex graph,

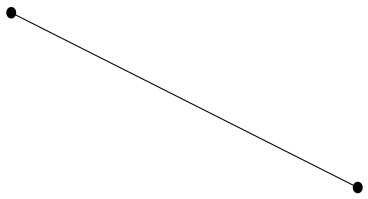
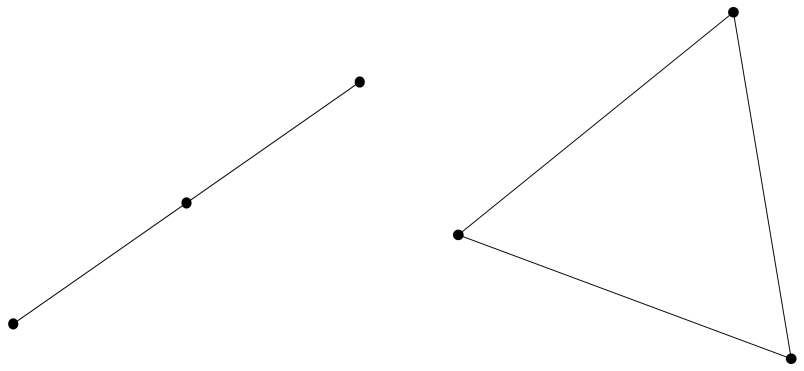


Figure A.1: Two vertex graphs



(a) (b)

Figure A.2: Three vertex graphs

the stable shapes that emerge will always be the same – albeit at different points in the plane or rotated. This property of the layouts speaks to the singular nature of system our algorithm operates on – the Laplacian matrix of the graph (see §3.4).

A.2 Three Vertex Graphs

The three vertex graphs are shown in Figure A.2. The struts ensure that a linear configuration is straightened and that a triangular configuration results in an equilateral triangle; clearly the edge lengths are uniform in these examples.

A.3 Four Vertex Graphs

In the four vertex graphs, shown in Figure A.3, the graph shown in Figure A.3(c) exhibits non-uniform edge length as a consequence of the additional struts and the strength of those strut ends in relation to the regular edge-based struts. In this case, the added repulsion struts are stronger than the edge struts by a factor of two.

The square and complete graphs form as expected with even edge lengths, but the five edge configuration in Figure A.3(e) is surprising, appearing as two triangles instead of a square with a diagonal edge. This result, however, has uniform edge length and is, therefore a more optimal solution, as all of the forces involved are evenly balanced.

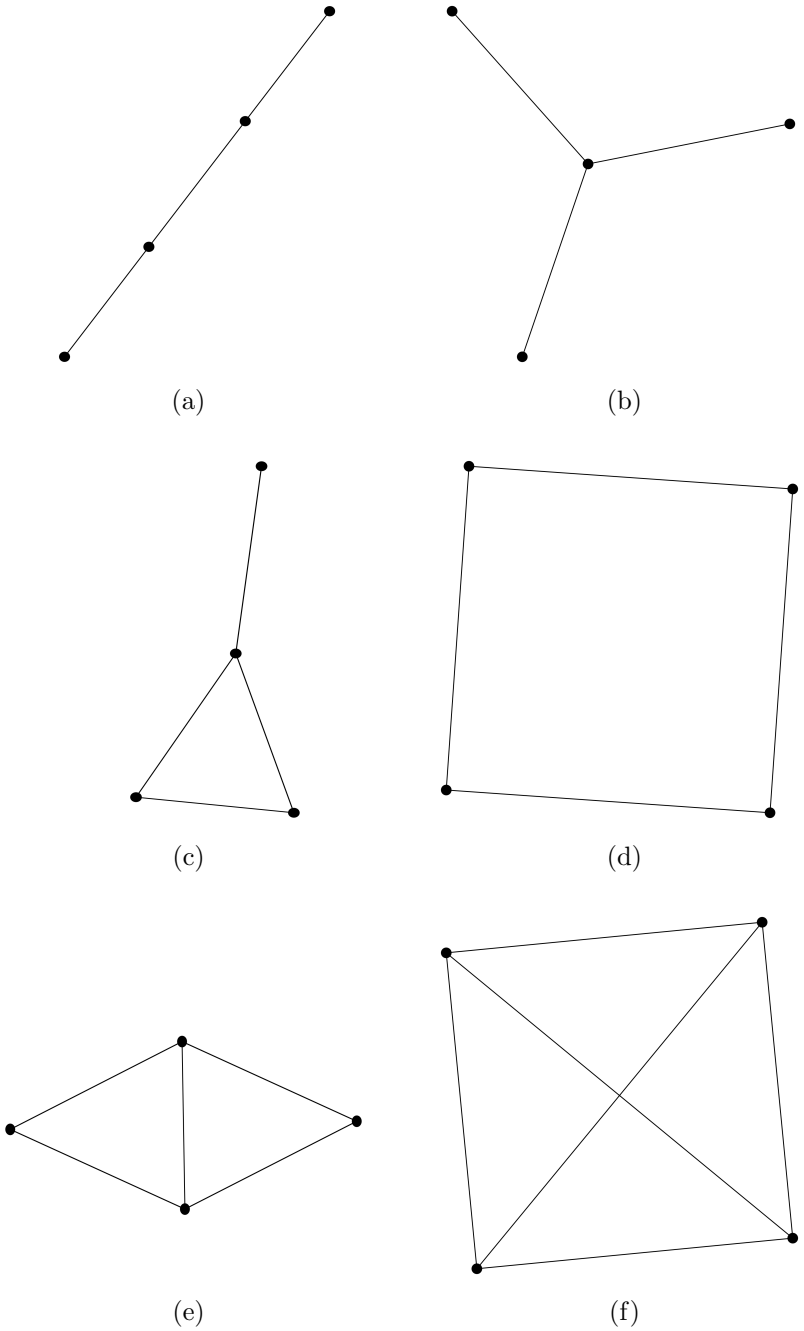


Figure A.3: Four vertex graphs

References

[1] graphdrawing.org. `http://graphdrawing.org/`.

[2] Walshaw graph repository. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition`, March 2008.

[3] Daniel Archambault, Tamara Munzner, and David Auber. Topolayout: Multilevel graph layout by topological features. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):305–317, March/April 2007.

[4] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[5] Giuseppe Di Battista, Roberto Tamassia, Peter Eades, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, June 1994.

[6] Eric Bonabeau. Graph multidimensional scaling with self-organizing maps. *Information Sciences—Informatics and Computer Science: An International Journal*, 143(1-4):159–180, 2002.

[7] Eric Bonabeau and Florian Hénaux. Self-organizing maps for drawing large graphs. *Information Processing Letters*, 67(4):177–184, August 1998.

[8] John Bovey and Peter Rodgers. A Method for Testing Graph Visualizations Using Games. In *Visualization and Data Analysis 2007*, volume 6495 of *Proceedings Electronic Imaging*. SPIE, January 2007.

[9] Brian Bradie. *A Friendly Introduction to Numerical Analysis*. Pearson Education Inc., 2006.

[10] Franz J. Brandenburg, Michael Himsolt, and Christoph Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Graph Drawing*, volume 1027/1996 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin / Heidelberg, 1996.

[11] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics (SIAM), second edition, 2000.

[12] M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz. The open graph drawing framework. Poster, 15th International Symposium on Graph Drawing, 2007.

[13] Markus Chimani and Carsten Gutwenger. Non-planar core reduction of graphs. *Discrete Mathematics*, 309(7):1838–1855, April 2009.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hilll, 1999.

[15] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

[16] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[17] David Forrester, Stephen G. Kobourov, Armand Navabi, Kevin Wampler, and Gary V. Yee. Graphael: A system for generalized force-directed layouts. In *Graph Drawing*, volume 3383/2005 of *Lecture Notes in Computer Science*, pages 454–464. Springer Berlin / Heidelberg, 2005.

[18] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 388–403, 1994.

[19] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–64, November 1991.

[20] Pawel Gajer, Michael T. Goodrich, and Stephen G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *8th Symposium on Graph Drawing*, Lecture Notes in Computer Science, pages 211–221. Springer Vërlag, 1999.

[21] Pawel Gajer and Stephen G. Kobourov. Grip: Graph drawing with intelligent placement. In *8th Symposium on Graph Drawing*, Lecture Notes in Computer Science, pages 222–228. Springer Vërlag, 1999.

[22] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–38, March 1986.

[23] Jörn Garbers, Hans Jürgen Prömel, and Angelika Steger. Finding clusters in vlsi circuits. In *Computer-Aided Design*, pages 520–523. IEEE, 1990.

[24] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Lecture Notes in Computer Science*, volume 3383/2005, pages 285–295. Springer-Verlag, 2005.

[25] Stefan Hachul and Michael Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing*, Lecture Notes in Computer Science, pages 235–250. Springer, 2005.

[26] Stefan Hachul and Michael Jünger. Large-graph layout with the fast multipole multilevel method. 2005.

[27] Stefan Hachul and Michael Jünger. Large-graphs layout algorithm at work: An experimental study. *Journal of Graph Algorithms and Applications*, 11(2):345–369, 2007.

[28] Ronny Hadany and David Harel. A multi-scale algorithm for drawing graphs nicely. Technical Report MCS99-01, Weizmann Institute of Science, 1999.

[29] Ronny Hadany and David Harel. A multi-scale algorithm for drawing graphs nicely. In *Graph-Theoretic Concepts in Computer Science*, volume 1665/1999 of *Lecture Notes in Computer Science*, pages 262–277. Springer Berlin / Heidelberg, 1999.

[30] Kenneth M. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, November 1970.

[31] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6(3):179–202, 2002.

[32] David Harel and Yehuda Koren. Graph drawing by high-dimensional embedding. *Journal of Graph Algorithms and Applications*, 8(2):195–214, 2004.

[33] David Harel and Meir Sardas. Randomized graph drawing with heavy-duty preprocessing. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 19–33, New York, NY, USA, 1994. ACM Press.

[34] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1995. ACM Special Interst Group on Computer Architecture, ACM Press.

[35] Yifan Hu. Efficient and high quality force-directed graph drawing. *The Mathematica Journal*, 10:37–71, 2005.

[36] Weidong Huang and Peter Eades. How people read graphs. In *PVis '05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation*, volume 109 of *ACM International Conference Proceedings Series*, pages 51–8. Australian Computer Society, Inc., 2005.

[37] Weidong Huang, Seok-Hee Hong, and Peter Eades. Predicting graph reading performance: a cognitive approach. In *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation*, volume 60 of *ACM International Conference Proceeding Series*, pages 207–16. Australian Computer Society, Inc., 2006.

[38] David S. Johnson. The np-completeness column: An ongoing guide. *Journal of Algorithms*, 3(1):89–99, 1982.

[39] Byong-Hyon Ju and Kyungsook Han. Complexity management in visualizing protein interaction networks. *Bioinformatics*, 19:177–179, 2003.

[40] Tomihisa Kamada and Satoru Kawai. An algorithm or drawing general unidirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

[41] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs: Methods and Models*. Lecture Notes in Computer Science. Springer-Verlag, London, UK, 2001.

[42] René Keller, Claudia M. Eckert, and P. John Clarkson. Matrices or node-link diagrams: which visual representation is better for visualising connectivity models? *Information Visualization*, 5(1):62–76, March 2006.

[43] Donald E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6(9):555–563, September 1963.

[44] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, third edition, 1997.

[45] Yehuda Koren. Graph drawing by subspace optimization. In *Proceedings of 6th Joint Eurographics - IEEE TCVG Symp. Visualization (VisSym '04)*, pages 65–74. Eurographics, 2004.

[46] Yehuda Koren, Liran Carmel, and David Harel. Ace: A fast multiscale eigenvector computation for drawing huge graphs. Technical Report MCS01-17, The Weizmann Institute of Science, 2002.

[47] Yehuda Koren, Liran Carmel, and David Harel. Ace: A fast multiscale eigenvector computation for drawing huge graphs. *IEEE Symposium on Information Visualization*, page 137, 2002.

[48] R. J. Lipton, S. C. North, and J. S. Sandberg. A method for drawing graphs. In *SCG '85: Proceedings of the first annual symposium on Computational geometry*, pages 153–160, New York, NY, USA, 1985. ACM Press.

[49] M. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software Practice and Experience*, 31(8):739–756, 2001.

[50] Bernd Meyer. Self-orgranizing graphs: A neural network perspective of graph layout. *Lecture Notes in Computer Science*, 1547:246–262, 19998.

[51] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[52] Andreas Noack. Energy models for drawing clustered small world graphs. Technical report, Brandenburg University of Technology at Cottbus, July 2003.

[53] Andreas Noack. An energy model for visual graph clustering. In G. Liotta, editor, *11th International Symposium on Graph Drawing (GD 2003)*, number 2912 in Lecture Notes in Computer Science, pages 425–436. Springer-Verlag, 2004.

[54] Andreas Noack. Energy models for graph clustering. 11(2):453–480, 2007.

[55] Emanuel G. Noik. Challenges in graph-based relational data visualization. In John Botsford, Arthur Ryman, Jacob Slonim, and David Taylor, editors, *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, volume 1, pages 259–277. IBM Canada Ltd. Laboratory Centre for Advanced Studies, IBM Press, 1992.

[56] H. C. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers*, 13(2):147–162, December 2000.

[57] H. C. Purchase, R. F. Cohen, and M. I. James. An experimental study of the basis for graph drawing algorithms. *Journal of Experimental Algorithmics*, 2, 1997.

[58] Mary Lynn Reed. Simulating small-world networks. *Dr. Dobb's Journal*, pages 18–23, April 2004.

[59] Peter Rodgers. Graph drawing techniques for geographic visualization. In Alan MacEachren, Menno-Jan Kraak, and Jason Dykes, editors, *Exploring geovisualization*, pages 143–158. Pergamon, December 2004.

[60] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, fourth edition, 1999.

[61] Andrew Smedley. Implementing a stochastic force-directed graph drawing algorithm: A software engineering report. Technical report, Lakehead University, April 2002.

[62] Roberto Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, April 1998.

[63] Martyn Taylor and Peter Rodgers. Applying graphical design techniques to graph visualisation. In *Proceedings of the 9th International Conference on Information Visualisation (IV05)*, pages 651–656. IEEE Computer Society, July 2005.

[64] Elizabeth Thomson. Team maps huge math structure. *Tech Talk*, 51(22):1,4–5, March 2007.

[65] W. T. Tutte. How to draw a graph. In *Proceedings of the London Mathematical Society*, volume s3-13, pages 743–767, 1963.

[66] Todd L. Veldhuizen. Dynamic multilevel graph visualization. *Computing Research Repository (CoRR)*, 2007.

[67] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 2nd edition, 1996.

[68] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. Tech. Rep. 00/IM/60, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, April 2000.

[69] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Graph Drawing, 8th Intl. Symp. GD 2000*, volume 1984 of *LNCS*, pages 171–182. Springer, Berlin, 2001.

[70] Eric W. Weisstein. k-connected graph. `http://mathworld.wolfram.com/k-ConnectedGraph.html`.

[71] Kay C. Wiese and Christina Eicher. Graph drawing tools for bioinformatics research: An overview. *Computer-Based Medical Systems*, pages 653–658, 2006.